

MUSS

USER MANUAL

VOLUME 1

This Manual gives the specification of the basic facilities of MUSS. Two other Volumes give information on the use of high-level languages and utilities respectively.

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUE 12

DRAFT: October 2019

The *MUSS User Manuals* were published in three volumes:

MUSS USER MANUAL VOLUME 1: Basic Facilities of MUSS

MUSS LANGUAGE MANUAL VOLUME 2: High-level Languages

MUSS UTILITIES MANUAL VOLUME 3: Utilities

This document is a reprinting of: *MUSS User Manual Vol. 1 Issue 12, 1886* published by the University of Manchester Institute of Science and Technology (UMIST). The principal author is Professor Derrick Morris. Morris, a Ph.D. student of Tony Brooker, was from 1964 in the Department of Computer Science, University of Manchester, but moved to UMIST in 1985. Tragically and entirely unexpectedly Derrick Morris died on 3rd of January 1997. UMIST, which separated from the University in 1994, merged back into the University in 2004. Its history is discussed at:

www.manchester.ac.uk/discover/history-heritage/history/umist.)

MUSS is the operating system developed initially at Manchester University and subsequently at UMIST. It was designed to be machine independent. It was initially used on the Manchester University computers starting with MU5 which was built after the Atlas system and for other computers which were acquired for use in the Computer Science Department. These computer systems included MU5, MU6, PDP-11's and RTV systems provided by Sperry Univac. In addition, the system was used by a number of organizations under license. The development of the MU5 System is described in [1] and in [2] *The MU5 Computer System*. In addition, the References given in [2] on page 258 provide a list of relevant publications up to that date. A description of the Machine-Independent Operating System (MUSS) as developed specifically for the Manchester systems is given in [3]. A course on modern Operating System Design, with an emphasis on machine-independence is given in [4]. A useful description of the MUSS compiling system is given in [5]. A full description of MU6 has not been published. The people involved were concerned about proprietary problems which arose during the implementation of MU5. An outline description of the hardware was given by D. B. G. Edwards, A. E. Knowles and J. V. Woods in [6].

Derrick Morris led the development of the MUSS Software System. Other main authors included Colin Theaker and Roger Phillips. The work that is described here was carried out over an extended period which originated in the Department of Computer Science and based on their long history of major contributions to computer systems.

In this edition of the documentation the RTV system is generally used as that was the main system in use at UMIST at that time. RTV was a research vehicle designed and built by Sperry Univac under contract with UM and UMIST. The RTV system was a VME-bus based system using Motorola CPUs. Early systems used the MC68000. But this was replaced by the MC68010 in order to more efficiently support paging. The hardware was designed to support the MC68020 when it became available. Sperry Univac was the main development partner with Motorola for the MC680xx from the time when Univac convinced Motorola to provide paging capability in the MC68010. Within Sperry UNIVAC, I (as Director of Research) initiated the collaboration of Univac and Motorola. And, specifically, I was able to convince Motorola that it was appropriate to provide paging hardware in addition to the segment structure as recommended by Bell Labs.

MUSS was from its conception “machine independent.” It could operate efficiently on hardware that did not directly support paging by using segments at the hardware level. This was done for the MC68000 systems among others.

This documentation mentions configuration specifics for MU6, VAX, PDP 11, MC680xx and indicates configuration requirements for other architectures. In addition, the documentation is an integral part of the system.

This text was created by OCR conversion of an original printed copy of *MUSS User Manual Vol. 1 Issue 12*, followed by extensive editing. For the most part the OCR was quite good. However, there were numerous errors and formatting problems. I have attempted to accurately reproduce the original text. Only obvious spelling or typographical errors were corrected. In any case very few typos were found in the original text and none were serious. There were a few “off by one” index errors, all quite obvious. I left the spelling checker set to UK English.

One difficult item was Appendix II which shows the various fonts that are available. A few entries are not exactly represented in L^AT_EX, but the codes are correct and the symbols are at least close.

If errors are found it may be helpful to refer to the scanned PDF of the original *MUSS User Manual Vol. 1 Issue 12, 1986* which can be found at: [MUSS-User-Manual-Vol-1-1986-orig-scan.pdf](#)

Any errors found should be reported to me at: michaeldgodfrey@gmail.com

October 13, 2019.

Bibliography

- [1] Ibbett, R. N. and Capon, P. C.: The Development of the MU5 Computer System. *Computer Systems, Comm ACM* Number 1, 13–24 (1978)
- [2] Morris, D. and Ibbett, R. N.: *The MU5 Computer System*. Macmillan Press, London (1979)
- [3] Morris, D. and Frank, G. R. and Theaker, C. J.: Machine-Independent Operating Systems. vol. Information Processing 77, 819–825. IFIP, North-Holland Publishing Co. (1977)
- [4] Theaker, C. J. and Brookes, G. R.: *A Practical Course on Operating Systems*. MacMillan Computer Science Series. MACMILLAN, London, UK (1983). Pgs. 1–203
- [5] Barringer, H. and Capon, P. C. and Phillips, R.: The Portable Compiling System of MUSS. *Software–Practice and Experience* 9, 645–655 (1979)
- [6] Edwards, D. B. G., and Knowles, A. E., and Woods, J. V.: MU6G: A new design to obtain mainframe performance from a mini-sized computer. *ACM 7th Annual Symposium on Computer Architecture* (1980)

Contents

USER MANUAL	i
DRAFT: October 2019	iii
Bibliography	vii
Contents	ix
1 INTRODUCTION TO THE SYSTEM	1
1.1 PROCESSES	1
1.2 VIRTUAL MACHINES	2
1.3 MESSAGES	3
1.4 SUPERVISORS	4
1.5 THE SYSTEM LIBRARY	4
1.6 PRIMER FOR THE MENU DRIVEN INTERFACE	5
1.6.1 Getting Started	6
1.6.2 Exercise 1	7
1.6.3 Editing	10
1.6.4 Backups, Archiving, Renaming and Deletion	13
1.6.5 Operations on Documents	15
1.6.6 Exercise 2	16
1.6.7 Operations on Programs	17
1.6.8 Program Development Facilities	19
1.6.9 Program Performance Measurement and Tuning	24
1.6.10 The Piece-wise Refinement of Programs	25
1.6.11 Exercise 3	25
1.6.12 Macro Documents	27
2 USING THE SYSTEM	31
2.1 INTERACTIVE OPERATION	31
2.1.1 Terminal Operation	31

2.1.2	Logging in	32
2.1.3	Break Actions	33
2.2	BATCH OPERATION	34
2.3	JOB CONTROL COMMANDS	34
2.3.1	Basic Command Format	34
2.3.2	Command Abbreviations	35
2.3.3	Parameters	35
2.3.4	Parameter Types	36
2.3.5	Sets of File Names as Parameters	39
2.4	USING FILES AND DOCUMENTS	41
2.4.1	File and Directory Names	41
2.4.2	Document Names	42
2.4.3	The Current File	42
2.5	COMPILING AND RUNNING PROGRAMS	43
2.5.1	Commands for job sequencing, etc.	47
2.6	A JOB CONTROL EXAMPLE	48
3	I/O STREAM MANAGEMENT	55
3.1	INTRODUCTION	55
3.2	CHARACTERISTICS OF INPUT AND OUTPUT STREAMS	56
3.2.1	Structure of an Input/Output Stream	56
3.2.2	Sectioned Streams	56
3.2.3	Message Streams – Messages and Synchronisation . .	56
3.2.4	IO Streams	57
3.2.5	Stream Modes	58
3.3	ASSIGNING DOCUMENTS TO STREAMS	61
3.3.1	Procedures for Defining and Releasing Streams . . .	62
4	BASIC INPUT/OUTPUT OPERATIONS	75
4.1	PROCEDURES FOR STREAM SELECTION, ENQUIRY ETC	75
4.2	BASIC INPUT/OUTPUT OPERATIONS	81
4.2.1	Character Input/Output Procedures	82
4.2.2	Record Input/Output Procedures	85
4.3	OTHER (CHARACTER STREAM) INPUT/OUTPUT PRO- CEDURES	85
4.3.1	Character Stream Input Procedures	85
4.3.2	Character Stream Output Procedures	87
5	FILES AND EDITING	93

5.1	GENERAL DESCRIPTION OF THE FILE SYSTEM	93
5.2	SHARING FILES	94
5.2.1	Network File Systems (not applicable to stand alone systems)	94
5.3	FILE SYSTEM COMMANDS	94
5.3.1	Other File and Current File Commands	100
5.3.2	File Manager Processes	103
5.4	FILE EDITING	105
5.4.1	Screen Editing Mode	106
5.4.2	Line Editing Mode	107
6	TERMINAL INPUT/OUTPUT	121
6.1	LOW LEVEL TERMINAL OPERATIONS	121
6.2	QUESTIONNAIRE AND MENU FUNCTIONS	124
7	LIBRARIES	129
7.1	LIBRARY SEGMENTS	129
7.2	THE SYSTEM LIBRARY	130
7.3	PRIVATE LIBRARIES	130
7.4	PROCEDURES FOR CONSTRUCTING LIBRARIES	131
7.5	PROCEDURES FOR IMPLEMENTING CALLS ON LIBRARY PROCEDURES	134
7.6	INTERPRETIVE CALLING OF LIBRARY PROCEDURES	136
8	OPERATOR FACILITIES	139
8.1	BOOTSTRAPPING	139
8.2	CONFIGURATION	139
8.2.1	Terminology	140
8.2.2	Configuration Commands	141
8.2.3	Configuration File	145
8.3	ACCOUNTING AND USER MANAGEMENT	147
8.3.1	Commands	148
8.3.2	Primitive Commands	148
8.3.3	High-level Commands	151
8.4	PERFORMANCE MEASUREMENT	152
8.4.1	Commands	152
8.4.2	Organisational Commands	153
8.4.3	High-Level Procedures	153
8.5	MISCELLANEOUS COMMANDS	155

9	SYSTEM PROGRAMMING LANGUAGE	161
9.1	INTRODUCTION	161
9.1.1	Modular Structure	162
9.1.2	Module Linking	162
9.1.3	The Storage Model	162
9.1.4	Initialisation	163
9.1.5	Modes of Compilation	163
9.1.6	The Command Structure of a Typical Compile Job	163
9.1.7	The Format of a Module	164
9.1.8	Kinds of Modules	166
9.2	KINDS OF STATEMENTS AND SCOPE RULES	167
9.2.1	Kinds of Statement	167
9.2.2	Declaratives	168
9.2.3	Imperatives	168
9.2.4	Directives	168
9.2.5	Procedure Definitions	169
9.2.6	Blocks	169
9.2.7	Scope and Block Structure	170
9.2.8	‘Statements’ as the Compiler sees them	172
9.3	BASIC ELEMENTS OF THE LANGUAGE	173
9.3.1	Symbols	173
9.3.2	Statement Separators	174
9.3.3	Comments	174
9.3.4	Names	174
9.3.5	Integer Constants	174
9.3.6	Character Strings	177
9.3.7	Real Constant	177
9.4	DIRECTIVE STATEMENTS	177
9.4.1	Areas	178
9.4.2	Compiler Output	179
9.4.3	Initialisation	179
9.4.4	Binary Patching	180
9.4.5	MUSS Library Calls	180
9.4.6	Vstore Type	181
9.5	DECLARATIVE STATEMENTS	181
9.5.1	Label Declarations	182
9.5.2	Variable Declarations	182
9.5.3	Procedure Declarations	184
9.5.4	Literal Declarations	185
9.5.5	Data Vectors	187

9.5.6	Type Declarations	188
9.5.7	Examples of Variable/Type Declarations	189
9.5.8	Field Declarations	190
9.5.9	Space Declarations	190
9.5.10	V-store Declarations	191
9.5.11	Import Declarations	192
9.6	OPERAND FORMS	193
9.6.1	Variables	194
9.6.2	Constants	195
9.6.3	Field Declarations	195
9.6.4	Procedure Calls	196
9.6.5	Built-In Functions	196
9.7	IMPERATIVE STATEMENTS	201
9.7.1	Computations	202
9.7.2	Type	204
9.7.3	Conditions	207
9.7.4	Control Statements	208
9.7.5	Implicit Blocks	209
9.7.6	Examples of a Procedure Definition	209
10	PROGRAM DEVELOPMENT TOOLS	213
10.1	CLASSIFICATION OF ERRORS	213
10.2	PROCEDURES FOR TRAP HANDLING	215
10.2.1	Notes on the Handling of Errors and Traps	217
10.3	PROCEDURES FOR FAULT MONITORING	218
10.4	RUNTIME DIAGNOSTICS	219
10.4.1	Brief Summary of the Runtime Diagnostic Facilities	221
10.4.2	Breakpoint Commands	223
10.4.3	Profiling Commands	225
10.4.4	Diagnostic Enquiry Commands	227
10.4.5	Diagnostic Control Command	234
10.5	LOW LEVEL RUNTIME DIAGNOSTIC COMMANDS	234
11	DOCUMENTATION AIDS	237
11.1	WORD PROCESSING	237
11.1.1	The Text Facility	237
11.1.2	Basic Facilities of Text	238
11.1.3	Layout Control Facilities (@V)	241
11.1.4	Special Layout Effects	243
11.1.5	Summary of Text Commands	244

11.1.6	The Spell facility	245
11.1.7	Listing Facilities	246
11.2	FLOCODER	246
11.2.1	The TITLE Statement	250
11.2.2	The COL Statement	250
11.2.3	The ROW Statement	251
11.2.4	The FLOW Statement	251
11.2.5	The BOX Statement	252
11.2.6	The END Statement	252
11.2.7	Flocoder Procedure Specifications	253
12	GRAPHICAL FACILITIES	259
12.1	GKS LANGUAGE BINDING	259
12.1.1	Control Function Specifications	261
12.1.2	Segment Attributes	261
12.1.3	Segment Manipulation Functions	262
12.1.4	Output Functions	262
12.1.5	Initialisation of Input Devices	262
12.1.6	Setting Mode of Input Devices	263
12.1.7	Request Input Function	263
12.1.8	Sample Input Functions	263
12.1.9	Event Input Functions	264
12.1.10	Workstation Independent Primitive Attributes	264
12.1.11	Normalisation Transformation	265
12.1.12	Workstation Transformation	265
12.1.13	Utility Functions	265
12.1.14	Inquiry Functions for Operating State Value	266
12.1.15	Inquiry Functions for GKS Description Table	266
12.1.16	Inquiry Functions for GKS State List	266
12.1.17	Inquiry Functions for Workstations	267
12.1.18	Pixel Inquiries	268
12.1.19	Inquiry Functions for Workstation State List	268
12.1.20	Inquiry Functions for Workstations Description Table	270
12.2	USING GKS FROM FORTRAN	271
12.3	USING GKS FROM PASCAL	273
12.4	GKS WORKSTATIONS	276
12.5	GKS ERROR MESSAGES	276
12.5.1	States	276
12.5.2	Workstations	277
12.5.3	Transformations	277

12.5.4	Output Attributes	278
12.5.5	Output Primitives	279
12.5.6	Segments	279
12.5.7	Input	279
12.5.8	Metafiles	280
12.5.9	Escape	280
12.5.10	Implementation Dependent	280
12.6	RASTER SCAN DEVICE FACILITIES	281
12.6.1	The Laser Process	287
13	VIRTUAL STORE MANAGEMENT	293
13.1	COMMANDS	294
14	CREATION AND CONTROL OF PROCESS	301
14.1	PROCESS CREATION	301
14.2	SCHEDULING	301
14.3	CPU TIME	301
14.4	COMMANDS	302
14.5	MULTI-TASKING WITHIN A PROCESS	305
15	INTER-PROCESS COMMUNICATIONS	309
15.1	MESSAGE COMMANDS	311
15.2	DIRECT COMMUNICATION COMMANDS	314
15.3	FORMAT OF MESSAGES	315
15.4	CONNECTION TO DEVICES	316
16	SEMAPHORES	319
16.1	COMMANDS	319
16.2	RELEASING SEMAPHORES	320
17	DISC/MAGNETIC TAPE FACILITIES	323
17.1	COMMAND PROCEDURES	323
17.1.1	Organisational Functions	324
17.1.2	Fixed Block Transfers	325
17.1.3	Variable Block Transfers	325
17.2	UTILITIES	326
18	COMPILER TARGET LANGUAGE (MUTL)	333
18.1	INTRODUCTION	333
18.1.1	Registers	337
18.1.2	Arithmetic precision	337

18.1.3	Instruction and Data Stores	337
18.1.4	Operand types and lengths	338
18.1.5	The CPU	339
18.2	OVERVIEW OF MUTL CONCEPTS	339
18.2.1	Compilation Units (Modules)	339
18.2.2	Storage control	340
18.2.3	Data types	340
18.2.4	Allocation of names	341
18.2.5	Code efficiency and optimisation	341
18.2.6	Program error detection and diagnostics	342
18.2.7	MUTL Code Output	342
18.3	TYPE DEFINITIONS	343
18.4	STORAGE MAPPING	348
18.5	STATIC DATA STORAGE DECLARATION	352
18.6	LITERAL DECLARATIONS	360
18.7	PROGRAM STRUCTURE DECLARATIONS	365
18.8	LABEL DECLARATION	371
18.9	PICTURE DECLARATIONS	373
18.10	CODE PLANTING	373
18.10.1	Operation Codes	374
18.10.2	Operands	384
18.10.3	Restrictions on the use of Operands	386
18.10.4	Mode Conversions	386
18.10.5	Mathematical Functions	390
18.11	MUTL INITIALISATION	398
18.12	TARGET MACHINE PARAMETERS	401
18.13	DIAGNOSTICS	403
19	COMPILER WRITER UTILITIES	409
19.1	SYNTAB	409
19.1.1	The Notation for describing syntax	409
19.1.2	The Syntactic Scan Procedure	413
19.1.3	Operation of the Scan Procedure	414
A1	APPENDIX 1 – CHARACTER CODES	419
A2	APPENDIX 2 – FAULT REASONS	427
	Index	433

MUSS

USER MANUAL

VOLUME 1

CHAPTER 1

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

1 INTRODUCTION TO THE SYSTEM

The Manchester University Software System (MUSS) comprises an operating system, compilers for the standard languages and some utilities. These include the Core and GKS (Chapter 12) graphics packages, text and graphics editors, a text formatter and software development tools (Chapters 10 and 11). Several styles of command language are allowed one of which provides a Unix¹-like environment. The whole system is implemented in a machine independent way and runs on a variety of computers. In the design of MUSS the emphasis was placed on providing an efficient and compact system, suitable for both single computer and network operation. At the same time, ease of modification and adaptability was an important objective, and it is expected that individual installations will modify and extend the system to suit their own needs. The description of MUSS given in this Manual is intended more for system programmers working on the system, than ordinary users. Only a small fraction of the material covered in this Volume needs to be known by an average user. In fact, the recommended way of using the system for straightforward applications is through a menu driven interface for which a short ‘primer’ style of introduction is given in Section 1.6. It may also be useful for a user to have a feel for the overall structure of the system and to know some of the terminology. This is very briefly covered in Sections 1.1 to 1.5.

There are two other Volumes to this Manual which describe respectively the use, under MUSS, of the standard high-level languages (MUSS Language Manual Vol. 2) and a variety of utilities (MUSS Utilities Manual Vol. 3).

1.1 PROCESSES

The central part of MUSS is the operating system kernel. Its job is to manage processes. From the user point of view the rest of the software in the machine can be regarded as consisting of a number of concurrent activities, or processes, performing tasks such as

- execution of a background user job,
- execution of an online user’s commands,
- management of exchangeable media.

¹Trademark of Bell labs

In principle each of these processes can be thought of as executing within its own dedicated computer but having a formal means of communicating with the other processes. However, it is a characteristic of many operating system activities, and user jobs, that they require the use of a processor for only relatively small amounts of time. The rest of the time is spent waiting for something to happen, for example, waiting for the lineprinter to finish printing the current line, for a user process to supply more output for printing, or for an on-line user to type the next line of input. Consequently, it is possible for all of the processes to share the use of a single processor. Thus one of the main functions of the kernel is to allocate the processor to processes as required, giving each process the impression that it has a complete machine to itself. This illusion will break down if many users simultaneously initiate actions requiring a large amount of processor time. Under these conditions the processor will be allocated to each process in turn for a limited ‘timeslice’. Thus the time to complete a given task will become significantly extended.

1.2 VIRTUAL MACHINES

The idea of creating the impression that each process has a machine to itself is formalised in the notion of a ‘virtual machine’ which consists of

- (i) A share of the available processor time. As far as the process is concerned, it appears to have a complete processor to itself and need not be concerned with the fact that it is shared with other processes.
- (ii) A share of the available store. In MUSS each process has a large virtual store which is divided up into logical units of information called segments. Some of the segments are private to the process and cannot be accessed by other processes unless shared access is specifically requested. Others are common to all processes and contain the main components of the system software. Generally, processes are prevented from modifying the information in these common segments. In most implementations of MUSS the segments are further divided into ‘pages’ which are automatically moved between the disc and the main working memories as required. Although the detail of the store organisation is to some extent a function of the hardware architecture, MUSS disguises this as far as possible. A typical organisation is illustrated by the Motorola 68000, where the address is treated as follows

SEGMENT NO	BYTE ADDRESS IN SEGMENT
6	18

Of the 64 segments which this allows, the upper 32 are treated as common and the lower half are private or shared.

- (iii) A library of procedures which any process may call. These occupy some of the common segments, and are permanently loaded in executable form, thus they are immediately available. They can be thought of as built-in operations, extending the instruction set provided by the actual processor.

An important feature of the virtual machine is that each process has the freedom to organise itself within this machine. Thus, processes need not be written in the knowledge that they will be sharing the computer with other processes. Input/output is dealt with by calling library procedures, which give each process the appearance of having its own set of input/output devices.

1.3 MESSAGES

The processes of an operating system are not in general independent of one another, but need to communicate. In MUSS, processes communicate with one another by sending and receiving messages (using built-in library procedures to do this).

Every process in the system has a unique name, by which the other processes (and users) may refer to it, and any process can send a message to any other by specifying its name. (In fact, an internal System Process Number (SPN) is used, but this can be found from the process name). A single message may contain any amount of information from a few bytes to a full segment.

The message system also forms the only means of communication between the process in a virtual machine and the outside world. The peripheral devices are controlled by pseudo processes called *device controllers*, which communicate with other processes via the message system. Thus, for example, information to be printed on a lineprinter must be sent as a message to 'LPT', the device controller in charge of the lineprinters.

1.4 SUPERVISORS

User jobs are introduced into the system by processes called supervisors, which are permanently resident in the machine. A supervisor is a process which services requests (in the form of messages) from users to start jobs. Its main function is to create and start new processes to execute these user jobs, but it can also exert some control over the execution of any process which it has created by use of the appropriate library procedures.

One supervisor permanently resident in the system is called JOB. Also, there is another called UNIX for the creation of jobs that require a Unix style command language. In addition, any other process may act as a supervisor to provide alternatives to the basic system facilities. The role of the basic supervisor is simply to create a virtual machine to run a user process. This process then interprets the commands which 'drive' the job. Normally these job control commands are interpreted as calls on library procedures which are to be made within the virtual machine running the process. Differing user needs are met by the variety of procedures available. Some procedures might in practice be interpreters of other job control languages. Alternatively, the job control procedures, like other library procedures, can be called from programs written in the high-level languages and complicated job control sequences involving conditional and repeated actions can be written in the standard high-level languages.

When a user at an on-line terminal activates a process by communicating with JOB an automatic entry is forced into a procedure which presents job control options to the user by means of menus. Use of the system through this mechanism is described in Section [1.6](#).

1.5 THE SYSTEM LIBRARY

Pre-loaded into every virtual machine created by MUSS is a set of library procedures. These provide the process with access to all of the facilities of the system, and include

- mathematical functions
- basic input/output procedures
- compilers
- editors
- job control procedures

JCL interpreters
operating system interface procedures
(for example 'SEND.MESSAGE').

The remaining chapters in this Manual are mainly concerned with the specification and use of these procedures.

The average user would not normally have contact with the operating system interface procedures. They are used by the supervisors, the basic input/output procedures which interface the users read and print commands into the message system, and the job control procedures. The job control procedures are mainly concerned with defining the environment in which programs are to run. This usually means defining the 'documents' which form the inputs and outputs of a program.

Consistency exists between user programs and users typing commands on terminals, hence either may access any of the preloaded set of procedures, either may use any library procedure and either can extend this set of procedures by introducing private libraries.

1.6 PRIMER FOR THE MENU DRIVEN INTERFACE

In order to use the system you should have a USERNAME and a PASSWORD. If you wish to use the system and you have not been allocated a USERNAME use the name HELP.

Start from a terminal which is not in use. Any such terminal which is connected to 'JOB' will respond to the operation of the RETURN key with an invitation to 'login'. This invitation takes the form

"USERNAME PASSWORD? This is ..."

The information given in the position of the ... identifies the machine to which your terminal is connected and it need not concern the beginner. The correct response to the login invitation is to type your USERNAME and PASSWORD then operate the RETURN key. Users may define their own arbitrary passwords and thereby gain some measure of protection for their personal files. However, initially each users password is set to 'X', and so long as it stays as this it need not be typed. In other words the system will assume a default 'X' when no password is given. After you have given a

correct user name and password (or no password in the case of it being 'X'), a menu will be placed on the screen from which you may choose your next action.

Associated with each terminal is a *cursor* which is a way of indicating a position on the screen. The representation of the cursor is not the same on all kinds of terminal, some will reverse colour in the region of the cursor, others may use a flashing invitation to login. Type your USER NAME, if you have one, otherwise type HELP. The machines response should be an initial menu of the underscore and so on, but you will soon learn to recognise the cursor. Actions are invoked by moving this cursor to the required cell of the menu by means of the 'up/down/right/left arrow keys' and operating the RETURN key. Initially, for a new user, the only relevant choice is to give a name for the first module of documents which is to be created in the permanent memory of the machine (its 'file store').

1.6.1 Getting Started

The initial menu presented after you successfully login will have a space allocated for module names. This will be empty until a module is added, and the cursor will be placed on the first (empty) module cell. All that you have to do in order to create a module is to type its name then operate the RETURN key. At this stage the module will of course be empty. Any module introduced in this way will remain in the system, stored as a file, and when a user logs in to the same user space on a future occasion it will appear in the menu as a selectable item.

Selection of a module means operating the RETURN key with the cursor pointing to a module name, even a newly introduced one. This will result in a new menu appearing which gives the names of the documents contained in the module. Another feature of these module menus relates to the operations which can be performed on the documents of the module. This will be mainly ignored for the present. However, it is the reason the module menus have a panel mentioning current global and default functions. In a new module they will all be set to Edit and this is the only function with which you need to be concerned for the present.

In order to introduce a document into a module place the cursor in a position allocated for the first document name and type a suitable name. If you are ready to begin typing the actual document operate the RETURN

key whilst the cursor is still at the position of the new document name cell, and you will invoke the Edit function.

1.6.2 Exercise 1

As a first exercise in using the system consider in detail the use of what has been described so far. To create a module containing a document. Step one is to locate an unused terminal issuing the invitation to login. Type your USER NAME, if you have one, otherwise type HELP. The machine's response should be an initial menu of the form:

```
Current Directory:  ROOT                AT 12:40:35 ON 10-MAR-86
=====
MENU FUNCTIONS:                        End Job
Current Root Directory:  Help          Password:
Archive Media Name:      Label:
**                        **
**                        **
**                        **
=====
Subdirectories:

Modules:(Select with the cursor, Use ↑A to archive, ↑B to Backup)
```

It will be instructive at this point to explore all the positions at which the cursor can be placed, and to think about their significance. Use the left arrow key repeatedly and the cursor will move back a cell at a time, from its initial position on the new module cell, until it arrives on the time cell from where it will not move back. Now retrace these movements forwards using the TAB key (or control I) to move from cell to cell and consider the purpose of each cell in turn. The time cell is editable so that it may be corrected if a machine has it wrong, but this is not the responsibility of inexperienced users. Note that this time is not continuously updated, it is reset each time the menu is rewritten to the screen. The first operation of TAB will move the cursor to the date cell and the same remarks apply to it. The next selectable cell is the 'End Job' cell. This is not editable but placing the cursor here and operating RETURN is the normal way of signing off so as to release the terminal for another user. The next two selectable cells are editable and they provide a facility to access another users file space. Placing this users USER NAME and PASSWORD into the two cells and operating RETURN will connect your process into his personal file space, almost as if you had

logged in using that name and password Documents: in the first instance. The next two selectable cells are also editable and they provide a facility for accessing file archiving media, such as floppy discs. The use of these cells will be described later. Following this are five editable cells, preceded by '**', which are intended for commands to be issued to the underlying operating system. Beginners should ignore this facility although it will be discussed again later.

The filespace of a user may be distributed between a root directory and a number of subdirectories. After logging in only the files and modules of the root directory are directly available, but if the filespace has any subdirectories the names of those at the first level of the subdirectory hierarchy will appear as selectable items beneath 'Subdirectories' in the menu. Placing the cursor on the name of a subdirectory and operating RETURN makes the filespace of the chosen subdirectory directly available. In order to return to the files and modules of the root directory the cursor should be placed on the 'Current Root Directory' and the RETURN key operated. So that a user always knows which file directory is current its name is displayed at the top of the menu. The menu also caters for the creation and deletion of subdirectories, but these facilities are for the more advanced users and are described elsewhere.

Beneath the cells concerned with subdirectories you will notice a cell with 'Modules:' in it, manipulation of this cell is concerned with archives and backups of modules and files, these facilities are described later.

With the cursor back at the new module cell, type in a module name, e.g. MODULE1. Any sequence of alphanumeric symbols (and .) could be used as a module name but it is better to choose a name which will be meaningful on a future occasions. Whilst the cursor is still in the new module cell operate the RETURN key. Now the display will change to the following.

```

Module:  MODULE1
=====
Sub-directories:
=====
Current repertoire (↑R): Edit
Global  repertoire (↑G): Edit
Default repertoire (↑F): Edit
EXIT    EDIT    OPEN :
=====
```

Documents:

=====

The cursor will be positioned on the first cell allocated for document names. Type a suitable name, for example 'DOCUMENT1', and a new menu will appear asking you to indicate the page type. The only suitable type at this time is 'text', and the cursor will be on this option already. Operate the RETURN key and the system will enter editing mode on the first (empty) page of your document, presenting you with the following display.

HELP:/MODULE1>DOCUMENT1

=====

The heading of this display serves to remind you of the document and page which is being edited. It is referred to as the document banner. The rest of the space on the screen is available to display the text on the page. However, so that we may describe the editing facilities separately let us assume that you do not wish to fill in any detail in this document just yet. To exit from the editor hold the control key down and type E. This generates a special character which we call control E and denote by ↑E.

There are several features relating to cursor movements which have not yet been mentioned. As an alternative to stepping through every cell, the up arrow and down arrow keys may be used to move a row at a time. When this is done the cursor might also move sideways since it will locate on the nearest cell of the previous or next line. The text in cells may also be changed, if for example, you wish to change the name of a document. Experiment with this since it is not possible to itemise all possibilities here. The basic principles are:

1. If the cursor is to the left of a name, each operation of the right arrow key will move it one character position to the right until it is at the right most edge of the name . One further use of this key will take the cursor straight to the rightmost edge of the next name.
2. The left arrow key works in exactly the reverse manner, that is only when it is at the left hand edge of the name does it move a cell at a time as in the above exercise.
3. With the cursor positioned somewhere in a name the action of typing

new characters will insert these characters.

4. Characters may be deleted by using the control D, control X and delete keys which will produce the same effect as in the Editor which is described below.

1.6.3 Editing

Most screen editors have so many facilities that it is difficult for the beginner to get started. Unfortunately the editor in this system also has that tendency. Even though an attempt has been made to limit the facilities, there are many 'essential requirements' which must be met. The best approach to learning how to use the editor is to acquire a working knowledge of a few basic facilities, practice using these, and then gradually increase the range of your working knowledge as the need arises. There are three kinds of actions which need to be understood.

First, there is the introduction of new text and this is done simply by typing it in. Do not use the return key unless you especially require a newline at that point, since the system will automatically turn the last space into a newline when a line is full. In the case of a program you will of course wish to force newlines in quite often, but this would not be the case in a letter or report.

A second kind of action is concerned with moving the cursor to a particular position on the screen so that changes can be made. There are several ways of doing this, the simplest of them being by means of the arrow keys which will move the cursor one character position at a time right/left/up/down. Try them out by typing some arbitrary text into a document and then moving the cursor around it. You will discover by this means that the computer knows the difference between an empty position on the screen and a position into which you have typed a space symbol, although you will not be able to see any difference on the screen. For example, if the cursor is moved up from the right hand side of a long line to a preceding shorter line it will be automatically moved left by the system until it is on the leftmost unfilled position. Since the position of the cursor is always visible we will not detail all the variants of this behaviour. If by the above means the cursor is placed on some existing text the action of typing new characters will cause them to be inserted at the cursor position, and they will not replace the existing characters.

The third kind of action is deleting characters from the screen. There are four ways of doing this which are all equally useful in different circumstances. The key marked delete (or DEL) on the keyboard will delete the character preceding the cursor position each time it is operated. This is the character most recently typed unless the cursor has been moved by an arrow key, and it is the most useful method if an error is noticed whilst typing. At each operation of this key, the cursor and the rest of the line (if any) will move back one position, hence a series of incorrectly typed characters starting with the rightmost may be deleted by repeated use of the DEL key. Sometimes it is required to delete characters positioned at or in front of the cursor and control D (which we will denote by ($\uparrow$ D)) has this effect, sliding back the rest of the line as it does so, with the cursor remaining in the same position. A variant of this is control W ($\uparrow$ W) which works like $\uparrow$ D except it will remove all characters up to a space or newline on each operation. This is particularly useful for deleting complete words. The 4th method is by using control X ($\uparrow$ X) which will delete all the line to the right of the cursor position including the newline character, thus bringing as much as will fit of the next line on to the current line and repeating the process on all subsequent lines to the end of the page. In fact this operation of compaction after deletion is slightly more complicated than this. The system distinguishes between newlines which it has generated because lines become full and those explicitly typed to establish a desired layout. It attempts to preserve the user enforced layout.

The last set of characters to be deleted are saved in a deletion buffer. These may subsequently be recovered using control R ($\uparrow$ R). The main uses of this are to remedy mistakes when characters have inadvertently been deleted, or to move strings around a document by deleting the characters to be moved. repositioning the cursor at the new location and typing $\uparrow$ R. The deleted characters may be recovered any number of times and so this is also an effective way of duplicating strings of text. Any number or combination of the deletion keys may be used together, providing no cursor repositioning operations are used in between.

Most of the other special actions of the editor are invoked by use of control codes. For example, control E ($\uparrow$ E) has already been mentioned as the way of terminating the editing operation. A complete list of control codes is given below, but there is no need to remember them. Any time during editing you may type 'escape' 'escape' and an abbreviated list of all the control code functions will be displayed. Actually the display will also provide a menu of

additional functions which is described later. The way to return from this state to normal editing is by typing $\uparrow E$.

Summary of the control code actions

$\uparrow B$	Search Backward for the string which follows
$\uparrow B$	Repeated will scroll half a page back
$\uparrow D$	Delete the current character
DEL	Delete previous character
$\uparrow E$	Exit from Edit mode
$\uparrow F$	Search Forward for the string which follows
$\uparrow F$	Repeated will scroll half a page forward
$\uparrow I$	Insert spaces up to the next tab position
$\uparrow K\}$	These commands are concerned with hierarchically structured documents and are further discussed in Section 1.6.10
$\uparrow L\}$	
$\uparrow N\}$	
$\uparrow O$	Recursive call of editor
$\uparrow P$	Invoke a formatting procedure
$\uparrow R$	Recover Deletion
$\uparrow T$	Duplicate line
$\uparrow U$	Move forward to next word
$\uparrow W$	Delete up to next space (1 word)
$\uparrow X$	Delete rest of line

The use of the string search commands probably requires further explanation. After either Control F or Control B, the editor accepts characters which it attempts to match in the text. The cursor is moved (forward for Control F, backward for Control B) until a match is found. The search is incremental in that as characters are typed the current string is used. Thus, Control F followed by A matches at the first A. If B is then typed the scan continues to the first occurrence of AB, and so on. Typing any Control character, except whichever of $\uparrow B$ or $\uparrow F$ was used initially, terminates the search and results in normal processing of the Control character.

A further use of $\uparrow F$ or $\uparrow B$ will result in the cursor moving to the next occurrence of the given string, and this may be repeated. If instead of typing a string after a $\uparrow F$ (or $\uparrow B$) a further $\uparrow F$ (or $\uparrow B$) is given the cursor will be moved ten lines forward (or back). This again may be repeated and is a way of moving in 'half screen' units through a large page.

The tab positions are also displayed on the ‘escape’ menu. Initially they are set for every eight characters, so typing a TAB on the keyboard has the effect of inserting spaces to position the cursor at the next tab position. This cell on the menu is alterable, so users may redefine the tab positions to suit their own layout requirements.

It is also possible to perform certain editing operations repeatedly throughout a page by selecting the ‘Repetition command’ cell on the ‘escape’ menu. Novice users are advised not to use this facility until they have gained some experience with the basic functions of the editor. The repetition command operates from the current cursor position on a page. The command which may be specified in this cell takes the form of a single letter accompanied by a delimited character string. The commands provided are

- A Position the cursor after the next occurrence the string
- B Position the cursor before the next occurrence of the string
- C Copy to the start of the next line containing the string
- D Delete the next occurrence of the string
- I Insert the string
- S Skip to the start of the next line containing the string.

For example, the command A/can/1/not/ inserts ‘not’ after every occurrence of ‘can’. The character / is used as the delimiter of the strings in this example, but it could equally well be any other character which does not appear in the strings.

At any stage in the repeated command, it is possible to verify the position of the cursor on the page and the effects of the editing operation by including a ? in the command. The current page is displayed and the user is expected to type one of three keys, as follows

- | | |
|--------|--|
| SPACE | continue performing the editing command |
| RETURN | ignore this occurrence of the string and restart the editing command |
| Other | abandon the repeated editing command. |

1.6.4 Backups, Archiving, Renaming and Deletion

In addition to selecting a module to edit, other facilities exist which operate on a module as a whole. When the cursor is pointing at a module name, the

menu cell containing the name is alterable and so it is possible to change the name of the module by editing the cell or delete it completely by clearing the cell.

There is also a facility for creating a copy of a module for security reasons. Two levels of security exist. The first is to produce a backup version of a module. This exists as a file on the main disc of the computer and may be retrieved at a later date if the module has accidentally been corrupted or deleted. To produce a backup copy, position the cursor on the module name and type control B ($\uparrow$ B). The second level of security is to produce an archive copy of the module on some removable media, such as a floppy disc. This is achieved by positioning the cursor on the module name and typing control A ($\uparrow$ A). This will only operate if the name and label of the floppy disc has been typed in the archiving media cells on the menu and the disc has been inserted in the drive.

To find the names of modules which have a backup version or which have an archive copy on the current floppy disc, the cursor should be positioned on the 'Modules:' cell and the RETURN key operated. When positioned on this cell, the display will cycle round the list of modules, backup versions, archived versions (if any) and other files (such as binary programs). When the backup modules are displayed, the modules may be deleted or renamed by modifying the name in the same way as with the original modules. An archive copy may also be produced by positioning the cursor on the name and typing $\uparrow$ A. To retrieve a module from its backup, the cursor should be positioned on the backup module name and $\uparrow$ R typed.

When the menu shows the archive versions of files, the only options available are to delete the file by clearing the name, or to recover the file by pointing at the name and typing $\uparrow$ R. Module names on the floppy disc have '.M' appended to them to distinguish the modules from other types of files. Similarly, backup versions when archived have the characters '.A' appended to the name.

The final menu displaying the names of all other types of files held in the user's directory is probably of least interest to the novice user. It displays the names of binary program files and simple text files and, as with the module names, these may be deleted or renamed by modifying the cell containing the name. An archive version may also be produced by pointing at the filename and typing $\uparrow$ R. If a text file is selected by positioning the cursor and typing

RETURN, the editor (described in Chapter 5) will be invoked to edit the file.

1.6.5 Operations on Documents

The creation and editing of documents should now be clear. In order to make other operations available we must return to reconsider the role of the middle panel on the module menu, on which discussion was previously deferred. The basic idea behind the system is to allow a *repertoire* of functions to be assembled which provides for each of the operations that a particular class of user needs, and excludes all others. For example, one class of user might be a secretary and the class of functions might be

```
EDIT
PRINT
CREATE MEMO
CREATE LETTER
INITIALISE FLOPPY
```

A table driven mechanism is provided which allows a document, prepared in the correct format, to be used to specify such a set of functions and the translation of each of them into a sequence of commands to be interpreted by the underlying operating system. This kind of document is called a function repertoire. There is no need to understand the detail encoding rules at this stage, but it will be helpful to have in mind its general form. The first page gives the list of functions available in the repertoire. This is followed by a pair of pages for each function. One of these is a 'prompting page' which in practice will indicate the defaults of certain parameters which the user might on some occasions need to change. It can be blank, and in these cases no prompting will occur. The second page contains the sequence of commands which will achieve the desired effect. They are often quite complicated and are best ignored except by the designer of the repertoire.

If a repertoire has been made the current repertoire of a module, by one of the means described below, the action of picking a document will result in the first page of the repertoire appearing on the screen. It will in fact be preceded by the document header so that there is a clear indication of the document which is selected and the functions which may be executed on it. A function is selected in the obvious way by moving the cursor to its name and operating RETURN. If the chosen function has no prompting

page the associated command sequence is invoked, otherwise the prompting page will appear on the screen so that any unsuitable default values may be changed. After this a final RETURN will invoke the associated command sequence.

1.6.6 Exercise 2

We will return later to consider the standard repertoires which are available to users of the system. Here we need to exercise the mechanism for associating repertoires with modules. Let us assume that a standard repertoire exists in the file space of the user called UTIL, which provides the above mentioned secretarial functions. Further that it is in a document called SEC.REP of the module S.REPS. Either by repeating exercise 1 or by continuing from where it ended, you should be able to create an empty document called DOCUMENT1 in MODULE1, and cause the menu for this module to be displayed.

In order to associate the SEC.REP repertoire with this module proceed as follows. First make it the 'global' repertoire. To do this type the name of the module containing the repertoire, in this case UTIL:S.REPS into the cell after 'OPEN :'. On operating the RETURN key the module menu for S.REPS will appear. This module menu will contain several documents all of which are repertoires. Place the cursor on the one you require (SEC.REP) and operate control G (↑G). The global repertoire cell on the menu will confirm your choice. Now type control E (↑E) and the menu for the first module will reappear with the global repertoire cell still set to SEC.REP. In order to make a permanent record of this global repertoire in your module, position the cursor on the global repertoire cell of the module menu and operate control F (↑F). The display should now look like the following.

Module: MODULE.1

```
=====
Sub-directories:

=====
Current repertoire (↑R): UTIL:S.REPS>SEC.REP
Global function      (↑G): UTIL:S.REPS>SEC.REP
Default function     (↑F): UTIL:S.REPS>SEC.REP
EXIT      EDIT      OPEN :
```

```
=====
Documents:  Document1
=====
```

Now complete this exercise by using the edit facilities to compose a document and print it by selecting the print option in S.REPS.

It is worth noting that if you inadvertently pick a wrong option from a menu then typing control E ($\uparrow$ E) will redisplay the previous menu.

1.6.7 Operations on Programs

Other standard repertoires exist to assist in the preparation, development, monitoring and refinement of programs. The one for Pascal programs is called PAS.REP. Minor variants of this repertoire exist for other languages such as Basic, Fortran and C. These repertoires are also in the module S.REPS and they can be associated with any user module into which program source is to be placed by proceeding as in Exercise 2. When a document of the module is selected with the PAS.REP repertoire operative the following display will be presented.

```
PAS.REP          Use control E ( $\uparrow$ E) to EXIT

EDIT              LIST

COMPILE and WAIT for log COMPILE

Provide BREAKPOINT and TRACE facilities

RUN

RUN and collect profile (or INSPECT profile)

JOBS / FILES / FLOPPY DISC status
```

This repertoire presents eleven options starting from EDIT, in connection with which there is only one thing to be added to what has been stated before. This relates to a facility to support the structuring of designs, where clear logical functions have been isolated and produced as separate documents. A macro facility exists so that the contents of a separate document may be inserted in the source text each time the program is compiled. It is also possible for the macros to have parameters, this and the use of documents for writing programs will be described later.

The functions of the PAS.REP repertoire are fairly self explanatory, but some will present the user with further choices. In the case of the LIST options the choice given relates to the destination of the listing. The default destination will be the printer closest to you, but you may substitute the name of another, if you have reason to do so, or the name of a file in the file store of the underlying operating system.

Two compile functions are provided, both when chosen will initiate a background job to compile a program. The difference between the two is that the 'COMPILE and WAIT' function waits for the compilation to complete and then invokes the editor to display the compilation record generated by the compiler. The COMPILE function allows the user to work at the terminal while the compile job runs in the background. For example, the user may browse through documents and modules but it is unwise to alter any documents actually being compiled. On completion of a compilation, a document containing the compilation record is inserted into the module. The insertion of a new document in this way is emphasised, whenever possible, by highlighting its name in the document list. For example, on some terminals the name will flash. The subsequent selection of a new document will turn off this highlighting. Compilation errors may be checked by using the editor to inspect the compilation record. For each compile error, the suspect line of Pascal will be printed together with the cause of the error and the pathname to locate the source line. The detail of the latter need not concern the user, since by placing the cursor at the start of the pathname and typing control O ($\uparrow$ O), a recursive call of the editor can be invoked, which results in the page containing the error being displayed with the cursor positioned at the faulty line. After correcting the fault in the source, typing $\uparrow$ E will end the edit of the faulty source document and the display will revert to the compilation record so that the user may consider the next compilation error. It may prove convenient to also edit this document so that it records which compile errors have been corrected.

The next four functions in the repertoire are all concerned with RUNning a previously compiled program. However, the RUN option is adequate for most purposes, and the other options will be described later.

It is probably beneficial at this point for you to try the facilities so far described to compile and run a program before the facilities for fault finding and performance measurement are considered.

1.6.8 Program Development Facilities

During the development cycle of software and later during its maintenance the correction of software faults can be tedious and time consuming. The facilities provided to assist fixing software faults fall broadly into two categories. Firstly, whenever a fatal program error occurs the state of the program can be inquired in source language terms. Secondly, the behaviour of a program can be monitored by inserting diagnostic checkpoints into the code of a program.

Consider first the facilities for inquiring the state of a program after it has encountered a fatal runtime error such as division by zero. The following will appear at the bottom of the screen when such a runtime error occurs:

```
Menu Options:  QUIT      RESUME      program / command execution
                DIAGNOSTIC INQUIRIES      BREAKPOINTS and TRACING
                POST MORTEM OUTPUT
```

This display gives the reason for the error and presents five choices of action. If the cause of the error is immediately obvious, for example the data file for the program was not created, then the QUIT option will abandon the execution of the program and redisplay the PAS.REP repertoire. More frequently, it will be necessary to investigate the state of the failed program. The 'DIAGNOSTIC INQUIRIES' option allows a user to do this interactively, while the 'POST MORTEM OUTPUT' option allows information about the program state to be printed or saved in a file for investigation later. Whenever one of these three menus is selected a further menu will be displayed and when the facilities it offers have been used operation of ↑E will cause the above error options to be redisplayed.

Selecting the 'DIAGNOSTIC INQUIRIES' option will display a menu similar to the one shown below.

```

=====
FUNCTIONS:  Display Procedures from      1   onwards
            Load Diagnostics for
=====

            Procedure Display
      No.    Name                      Source position
- >1      Real package                HELP: /EX.3>MAIN 1 12
      2      COMBINATION              HELP: /EX.3>MAIN 1 18
      3      Main program             HELP: /EX.3>MAIN 1 28
=====

      LOCAL VARIABLE      Inspection
ALL .                      .
                          .
                          .

**

```

This menu is in three parts. The central part of the menu displays information about the procedures of the program currently being executed. This display starts with the procedure in which the error occurred and is followed by the procedure which called it and so on for all the procedure frames on the program stack. Above the procedure display are some functions, and below it are a number of cells for inquiring the values of variables. At the bottom of the menu is an editable ‘**’ cell which allows experienced users to issue commands to the underlying operating system.

It is instructive at this point to run a faulty program to obtain the above menu, and then to move the cursor around to acquaint yourself with the selectable cells.

Generally, each line of the procedure display gives the name of the procedure, the position of their frame on the stack, and the position in the source of the statement currently being executed in that procedure. This display may include the names of system procedures but it is easy to recognise those procedures which belong to your program. Each line of the procedure display has two pickable cells. Selecting a ‘procedure name’ and operating RETURN nominates its variables as candidates for inspection. Selecting the ‘source position’ cell of a procedure will invoke the editor to display the program source with the cursor positioned on the statement currently being executed within the selected procedure. Alternatively, selecting the same cell with ↑B will display machine instructions in the vicinity of the

source statement currently being executed. Clearly this is a diagnostic aid for system programmers not beginners.

Once the variables of a procedure have been nominated for inspection, a variable's value is obtained by typing its name in one of the variable cells and operating RETURN. One of these cells contains 'ALL' and selecting it will output all the variables of the selected procedure. To modify variable values, first pick the 'Inspection' cell itself, this will change its contents to 'Modification', so that whenever the user subsequently requests variables a new value may be typed in.

The only aspects of this menu remaining to be discussed are the functions displayed at the top of the screen. The 'Display Procedures' function allows procedures further down the stack to be displayed, while the 'Load Diagnostics' function loads diagnostic information for system software. This latter function is for use by system experts.

As a continuation of the previous practical exercise, obtain the 'DIAGNOSTIC INQUIRIES' menu on the screen and then display the source and variables of the procedures mentioned in the menu.

Diagnosing the cause of software faults is best done interactively. However a printed post mortem of a program is useful for the investigation of faults when access to the computer is limited. Because some programs have very large data structures it can be wasteful and irritating for a post mortem to include all the variables of a program. Therefore the 'POST MORTEM OUTPUT' option of the ERROR menu presents two choices.

- 1) A default post mortem that outputs information about procedures with frames on the stack and outputs their variables, but some of the large data structures may not be output.
- 2) A post mortem where the user selects the variables to be output.

On selecting POST MORTEM OUTPUT a menu similar to the one below is displayed.

Post Mortem destination is LPT*

=====

FUNCTIONS : Output default post mortem and exit
 Output variables requested below
 Load Diagnostics for

=====

	Procedures from	1	onwards with diagnostics
1	Real package		ALL SCALAR variables
2	COMBINATION		ALL SCALAR variables
3	Main program		ALL SCALAR variables

=====

Libraries with diagnostics

This menu contains some functions, a display of procedures and a display of libraries. Initial users of the system can ignore this library display. Again it would be instructive at this point to display the above menu and move the cursor around to locate the cells of the menu. The menu is reasonably self explanatory but the output of a selective post mortem warrants some elaboration. Each line of the procedure (and library) display has a cell which specifies the variables to print. By selecting this cell repeatedly you will see the choices available. One such choice is to output all unstructured (scalar) variables, another choice is an empty cell in which the name of a variable can be typed. After setting the requests for variables, selecting the 'Output variables requested' function prints them. If further variables need printing then the above sequence of select and output can be repeated as often as required.

For most software faults the above facilities are perfectly adequate. However, it is sometimes difficult to locate some faults from the symptoms exhibited at the point of program failure. One reason for this is that the symptoms may be due to the interaction of several faults. For these situations there is a facility to insert breaks into a previously compiled program at strategically chosen points in a program. Execution of the program will then halt at these breakpoints and the state of the program can be investigated using the diagnostic facilities just described.

Breakpoints are inserted by selecting the 'BREAKPOINT and TRACE' option of the PAS.REP menu. This will display a menu similar to the one below that contains primarily a table of breakpoints currently in the program.

Display of BREAKPOINTS in Program
(Use ↑R to remove a breakpoint)

NEW Breakpoint

Procedure TRACE is OFF

A breakpoint is inserted by selecting the 'NEW Breakpoint' option which calls the editor to display the program source. Using the facilities of the editor the cursor is positioned on the statement where a breakpoint is required, then typing ↑G will insert the breakpoint. Further breakpoints, if required, may be inserted without leaving the editor. After inserting the necessary breakpoints ↑E will exit the editor and redisplay the table of breakpoints which will have been updated with the breakpoints just inserted. The exact position of a breakpoint can be confirmed by picking its entry in the table, this calls the editor which displays the relevant document with the cursor on the statement containing the break. A breakpoint is removed from the program by picking its entry in the table with ↑R. After setting the necessary breakpoints typing ↑E redisplay the PAS.REP menu and the RUN option can then be picked to start the program execution.

Whenever the program encounters a breakpoint during execution it displays the runtime error options previously mentioned. If you wish to investigate the program state then select the DIAGNOSTIC INQUIRIES option and proceed as described earlier. After this investigation selecting QUIT abandons program execution while RESUME resumes program execution from the breakpoint. If you wish to insert new or remove existing checkpoints then selecting the 'BREAKPOINT and TRACE' option will display a checkpoint menu to allow you to do this.

To help diagnose software faults the 'BREAKPOINT and TRACE' provides a facility to trace the sequence of procedures during a program's execution. If the procedure TRACE is enabled (ON) then when a program executes a line is displayed each time a procedure is called.

Sometimes it will be necessary to stop the execution of a program prematurely (i.e. break in), for example if the program is in a loop. To break into the execution of a program $\uparrow C$ is typed followed by a B in response to the ?? prompt.

In order to familiarise yourself with the checkpoint and trace facilities, insert some breakpoints into an actual program and then obtain a procedure trace for the program.

1.6.9 Program Performance Measurement and Tuning

When developing software applications that are CPU intensive, it is important to be aware of the execution performance required and the performance of software actually attained. Consider, for example, the performance demands for editing graphics interactively.

The principal facility for improving performance is a CPU profile that gives a breakdown of the CPU used by a program. A profile is obtained by the 'RUN and collect profile' option of the PAS.REP repertoire. This option runs the program and captures statistics about the programs behaviour, after which the 'INSPECT profile' option will create a document containing a profile from the captured statistics and invoke the editor to display it. This latter option allows you to choose two kinds of profile analysis, namely:

- 1) A breakdown of CPU used by the program on a per procedure basis,
- 2) A breakdown of CPU used by procedures on a per statement basis.

The profiling mechanism obtains statistics by sampling the program control register at periodic intervals during program execution. For programs and procedures that use a small amount of CPU time, the number of samples collected is often small, and as a consequence their profiles will be less accurate.

1.6.10 The Piece-wise Refinement of Programs

Now that the facilities for the creation of running, diagnosing and tuning programs have been discussed, it is appropriate to consider the facilities provided to support the piece-wise refinement of programs.

In fact these facilities are equally applicable to any form of document in which a hierarchical structure is desirable, for example a report. The basic idea is to remove detail from a document and replace it by a short text string which summarises its content and acts as a key by means of which the detail can be retrieved as necessary. In some cases creation of the detail will be postponed until the main form of a document has been established and this is consistent with the familiar concept of piece-wise refinement. This concept can be thought of as a page which has sub-pages. It can be applied recursively in that the sub-pages might themselves have sub-pages. A page (or sub-page) is the unit of text which may be screen edited. As described below the set of pages which make up a document form a tree structure whose topology relates to the logical structure of the document. Navigation around this tree structure from a terminal is accomplished with a very small number of keystrokes. Facilities also exist whereby detail initially dealt with in this manner can be substituted in line or detail which is initially in line can be taken out and referenced by a descriptive key. A simple example will make some of this clear.

1.6.11 Exercise 3

Consider a Pascal program to find the results of a quadratic function. This clearly has several logical parts which might be expressed thus

```
PROGRAM quads (input, output);
    VAR a, b, c, x1real, x2real, x1imag, x2imag: real;
    procedure to get the coefficients
    procedure to solve the quadratic equation
    procedure to show results
    BEGIN
        getcoefficients (a, b, c);
        solvequadratic (a, b, c, x1real, x2real, x1imag, x2imag);
        showresults (x1real, x2real, x1imag, x2imag)
    END.
```

If the above was input as a document the three 'pieces' which need refinement

might subsequently be introduced as sub-pages. Two steps are necessary to accomplish this. First the three pieces in question need to be marked as references to sub-pages. This is done by placing the cursor in front of each in turn and operating control N ($\uparrow$ N). The descriptive text for a sub-page is assumed to end at the next newline character, another sub-page or — and is not seen by the compilation system. The second stage involves causing the editor to recurse to the content of each sub-page by placing the cursor anywhere within the descriptive text referring to the sub-page and operating control O ($\uparrow$ O). Of course initially the page will be blank but each may be typed in the usual way. Subsequently, only the control O stage needs to be invoked in order to edit the sub-page. Examples of the sub-pages for the quadratic function might be

```

PROCEDURE get coefficients (VAR a: real; VAR b: real; VAR c: real);
BEGIN
    writeln ('Input a, b, and c values of the quadratic equation:');
    readln  (a, b, c)
END;

PROCEDURE solvequadratic (a, b, c:  real; VAR x1real:  real;
                        VAR x2real:  real; VAR x1imag:  real;
                        VAR x2imag:  real);
    VAR  discriminant:  real;
BEGIN
    discriminant := sqrt (b) - 4 * a * c;
    IF discriminant > 0 THEN
        BEGIN
            x1real := (-b + sqrt (discriminant)) / (2 * a);
            x2real := (-b - sqrt (discriminant)) / (2 * a);
            x1imag := 0;
            x2imag := 0
        END ELSE
            IF discriminant < 0 THEN
                BEGIN
                    x1real := -b / (2 * a);
                    x2real := x1real;
                    x1imag := (sqrt (-discriminant)) / (2 * a);
                    x2imag := -x1imag
                END ELSE
                    BEGIN

```

```
        x1real := -b / (2 * a);
        x2real := x1real;
        x1imag := 0;
        x2imaa := 0
    END
END;

PROCEDURE showresults (x1real, x2real, x1imag, x2imag: real);
BEGIN
    writeln ('x1 real      = ',x1real:10:5);
    writeln ('x2 real      = ',x2real:10:5);
    writeln ('x1 imaginary = ',x1imag:10:5);
    writeln ('x2 imaginary = ',x2imag:10:5)
END;
```

This program may now be run, try it.

There are occasions during the production of a document when the topology needs to be modified. For example, detail appearing within a page might need to be represented on a sub-page, or conversely, the contents of a sub-page might need to be substituted in line. This is achieved using control L ($\uparrow$ L) and control K ($\uparrow$ K) respectively. The creation of a sub-page containing some existing text is achieved by first deleting the text using the standard editor delete keys described earlier. The text saved in the deletion buffer, as would be recovered by typing $\uparrow$ R, may be inserted as a sub-page by typing $\uparrow$ L. The system uses the same mechanism for generating references to sub-pages and to the macro documents described in 1.6.12. Hence after typing $\uparrow$ L, a prompt for the name of a macro appears, and a blank name implies that a sub-page is required.

The converse action is to substitute a sub-page in place of its descriptive text. This may be achieved by placing the cursor within the text, as when editing it using $\uparrow$ O, and typing $\uparrow$ K.

1.6.12 Macro Documents

There is another mechanism provided by the editor similar to the sub-page described above. This allows documents which are accessible through the names in the module menu to be used as macros inside other documents. Instead of typing control N ($\uparrow$ N) to introduce a sub-page, a named document

may be referenced within the text simply by inserting the document name, preceded by the macro warning character `#`. For example, if the required macro document appears within the current module, a reference such as `#DOCUMENT1` could be used to cross-reference it. The document name might include a full path name, including the module name and the username for the file space holding the module. Thus, for example, a full reference might appear as `#HELP:MODULE.1>DOCUMENT1`.

The other control functions associated with sub-pages operate in a similar way for these macro documents. For example, to call the editor recursively, the cursor should be positioned at the start of the document name and `↑O` operated. As described earlier, a macro document may be defined using `↑L` and specifying a document name when prompted. The converse operation, using `↑K` to expand a sub-page in line, also functions with macro documents.

It is better in a big program to treat the main constituent parts this way. For example modules and procedures which have a clear functional interface and whose implementation may proceed independently of the rest. The sub-page mechanism is more appropriate for the more intimately connected sub-procedures and the bodies of complicated control constructs. A notable distinction between the two techniques is that macro documents might also include parameters. These are specified by including the parameters in parenthesis after the reference to the macro name. The parameters are treated as strings which are substituted in the macro body for the parameter references `#1`, `#2`, `#3`, etc. For example, `#MODULE.1>DOCUMENT1 (string1, string2)` refers to the document `DOCUMENT1` in `MODULE.1` with `string1` appearing in place of every reference to `#1` and `string2` appearing in place of `#2`.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 2

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

2 USING THE SYSTEM

This chapter gives an introduction to the operation of terminals and the use of the command language. More detailed descriptions of particular commands are given in later chapters. Because MUSS is primarily an interactive system, the emphasis in this chapter is on interactive operation, but all of the commands described can also be used in background jobs.

2.1 INTERACTIVE OPERATION

2.1.1 Terminal Operation

Normally the system only responds to input after a complete line including the carriage return or newline has been entered. It is also the norm for terminals to be in full duplex mode and for the terminal line to be configured so that the input is echoed. Screen editors will reconfigure the terminal lines so that each character is made available as it is read and automatic echoing of input is inhibited. After processing a line of input, the system usually responds with a prompt requesting further input. In general this prompt may be any sequence of characters, but a number of standard prompts are defined for certain common situations. These are:

- ?? The terminal is currently disconnected, and a valid connect command must be given before the terminal can be used for other purposes.
- ** The terminal is connected to a process which expects the next line to be a job control command (see [Section 2.3](#) below).
- This prompt is used in most cases other than the above, for example by an editor which is waiting for a command to be typed.
- >* This prompt results from an error in a process causing an entry into the trapping mechanism. Any job control command may be input at this time, as with **, but it is normally expected that the runtime diagnostic commands of [Chapter 10](#) will be used.

A further common convention is that when text is being input, for example during editing, the system prompts with the character which will terminate the text input.

When a terminal is disconnected, i.e. giving the ?? prompt, the user must connect it to the process with which he wishes to communicate. This is normally accomplished by typing a single character (M or P) to indicate the required mode of communication, followed by the process name. By this means a connection can be made to any process in the machine which has its channel 0 'open' (see Chapter 15). In some more complicated situations a process may be set up to make several channels accessible from a terminal. When a connection is to be made into such a channel, the channel number enclosed in brackets should follow the process name, for example, PROCESS10(3). However, most users will only need to connect with one of the standard supervisors, such as JOB, which will expect input on channel 0. The same function, but in addition when P is used the process named becomes the *primary* connection. A *primary* connection is remembered even though a terminal may subsequently be connected to another process, and the terminal reverts back to this primary connection at times when it would otherwise become disconnected, for example, when a process terminates.

At any time while a line is being input, the last character typed may be deleted by typing the backspace character (CONTROL H). This may be repeated as often as necessary to delete further characters, as far as the start of the line. Alternatively the cancel character (CONTROL X) may be used to delete the whole line.

When a terminal is connected to a process which is producing output at too fast a rate, the flow of output can be interrupted by typing the XOFF character (CONTROL S). The effect of this is to halt the output until an XON character (CONTROL Q) is input.

2.1.2 Logging in

In order to use the system in a general way a user must create a process to accept further commands, and the JOB supervisor is provided to make this easy. All it requires, once a connection has been made to JOB, is a single line of input giving

username password jobname options.

Where 'username' and 'password' identify the user who is logging in, 'jobname' is the name the user wishes to assign to his process. All of this information must be supplied, but the user may also optionally supply a time

limit in seconds (C), a terminal type (T) and a priority (P) for his job. If no priority is given the timesliced priority (31) will be used. Other priorities (which are not timesliced) should only be used for special purposes such as supervisors, or for background jobs.

Provided that the process creation information proves satisfactory, the terminal connection will be transferred to the newly created process which will respond by displaying a menu as described in Subsection 1.6.1. Commands may be typed into the cells containing ** and a selected command can be executed by placing the cursor on that cell and operating the return key. Alternatively the menu system can be abandoned by generating two consecutive breaks (See 2.1.3). In this case the screen will clear and a ** prompt will appear to indicate that the system is waiting for commands to be typed a line at a time. The command formats are the same in both cases.

A process is terminated by either the user or a running program calling the STOP command. This will resume the *primary* connection of the terminal.

A MUSS system will normally have other supervisors in addition to JOB. For example, the UNIX supervisor mentioned earlier (See 1.4).

2.1.3 Break Actions

Operation of the break key (or CONTROL C) is the means by which a process, or a connection, can be interrupted. The immediate effect is that a "???" prompt is given and the system awaits one further character to determine the required action as follows

return	means ignore the break
B	means force an interrupt in the connected process
D	means return to the primary or disconnected state
M	means establish a new interactive connection
p	means establish a new primary connection

If a terminal is removed from a process in this way it can be later reconnected providing the process name is known. Thus a user at one terminal may create a number of processes and switch the terminal connection between them.

2.2 BATCH OPERATION

There is very little distinction in MUSS between interactive and batch (or background) jobs. Mainly it is a difference in the way a job is scheduled depending on whether its priority is

```
9 - 11 meaning interactive
12 - 15 meaning batch.
```

Interactive jobs are timesliced and batch jobs are run one at a time in priority/arrival order provided that they are free to use the CPU. A further difference concerns the action after a fault trap is entered which is discussed further in [Chapter 10](#).

2.3 JOB CONTROL COMMANDS

After successful initiation through JOB, a process will be executing in the command interpreter and ready for the user to type commands which direct its execution. A prompt will be produced whenever it is expecting such a command.

Any of the procedures in the system library (or in some private user library, as described in [Chapter 7](#)) may be interpretively called as commands. The standard command interpreter is intended to meet the basic needs of sophisticated users. Other interactive command languages of a more ‘user friendly’ nature are implemented as macro processors which generate calls on the basic command interpreter (see [Chapter 6](#)). Alternatively any procedure which can be used as a command can also be called from a program written in one of the programming languages supported by the system. Thus complicated job control sequences can be written as programs to be compiled and executed.

2.3.1 Basic Command Format

A command is input as the name of the command to be called, followed by its parameters if any.

```
COMMAND PARAM1 PARAM2 ... PARAMn
```

The command name and its parameters are separated by spaces and the entire command is terminated by a newline. Rightmost parameters may be

omitted, in which case zero will be substituted; most of the commonly-used commands will assume a suitable default value in this case.

Some Examples of Commands are

LIST.FILE PROG LPT*	lists the file PROG on the lineprinter named LPT
LIST.FILE PROG	lists the file PROG on the terminal
EDIT filea fileb	prepares to edit a file 'filea' to produce a new file 'fileb'
PASCAL filep	calls the PASCAL compiler to compile the program in 'filep'.

In the command descriptions in this Manual, many of the command names contain the character '.', for example LIST.FILE. This is for readability only and is not actually required, and '.' typed anywhere in a command name will be ignored.

Optionally, the command name may be preceded by two asterisks

****LISTFILE PROG.**

This form is normally only used if commands are to be executed during compilation of a program; all of the system compilers recognise lines beginning with '**' and interpret such lines immediately as job control commands.

2.3.2 Command Abbreviations

Some of the command procedures in the system library have a unique two or three letter abbreviation which may be used instead of the full procedure name. For example, LISTFILE may be abbreviated to LF. The abbreviations are shown in the headings of the command descriptions given in this Manual, and in the INDEX, as underlined characters.

2.3.3 Parameters

Many of the procedures in the system library require parameters and return results. These may be almost any of the types allowed by the compiling system although the command interpreter does not deal with a few of the more unusual types.

Within the command descriptions of this Manual, the types of the parameters are indicated by the command heading using the type notation of MUSL (see Section 9.5.2), for example

LIST.FILE (ADDR [LOGICAL8], ADDR [LOGICAL8])

This means that LIST.FILE has two parameters, which are bounded pointers to vectors of bytes representing characters. In the command descriptions the parameters are referred to individually as P1, P2, etc.

Some command descriptions may also refer to a set of global variables PW0, PW1, etc. (of type ADDR) and PWW1, PWW2, etc. (of type LOGICAL64). These are used for passing results back from kernel procedures and are not usually of importance in job control contexts, with the exception of PW0 which returns status information on completion of a command. If PW0 = 0, the command completed successfully; otherwise PW0 contains a fault number. The fault numbers are listed in Appendix A2, but the command interpreter will normally output a suitable error message if such a fault is detected. If the 'PW' variables are to be accessed from a user program a suitable declaration must be given which identifies their position at the base of the stack segment. In MUSL notation (Chapter 9) this would be

```
*GLOBAL 5 :: AREA 5 MUST BE PLACED IN THE STACK SEGMENT
ADDR PW0, PW1, PW2, PW3, PW4, PW5, PW6;
LOGICAL64 PWW1, PWW2, PWW3;
```

2.3.4 Parameter Types

Job control parameters may be written in several different ways; according to the parameter type. The complete list of permitted types is

- 1) LOGICAL8
- 2) LOGICAL16
- 3) LOGICAL32
- 4) LOGICAL64
- 5) INTEGER8
- 6) INTEGER16
- 7) INTEGER32
- 8) REAL32
- 9) REAL64

- 10) ANY USER DEFINED TYPE
- 11) A POINTER TO ANY OF 1 – 10
 - e.g. ADDR LOGICAL8
 - ADDR LOGICAL16
- 12) A BOUNDED POINTER TO ANY OF 1 – 10
 - e.g. ADDR [LOGICAL8]
 - ADDR [LOGICAL16]

When a parameter specification gives only the arithmetic type without a specifying size (e.g. INTEGER) the default size may be different on different machines. The distinction between logical and integer only applies when type conversions are needed which involve a size change. Logicals are always zero extended on the left and integers are always sign extended on the left. Otherwise, apart from two special cases detected by the command interpreter, the entire group 1 – 7 may be thought of as integer parameters. The special cases are

LOGICAL8	which is assumed to represent a character
and LOGICAL64	which is assumed to represent a packed string of up to eight characters.

The forms in which the user may express the values of scalar parameters of types 1 – 9 are

- a) a signed integer constant e.g. 99, -1
- b) a signed real constant e.g. 99.175
- c) a character constant e.g. A, i, +
- d) a packed string (name constant) e.g. MU6G
- e) a hexadecimal constant e.g. %175AF
- f) a nested procedure call e.g. <COS .5 >.

The permitted forms of parameters are shown in the following table. (/ indicates the form is permitted)

	a	b	c	d	e	f
1			/		/	/
2	/				/	/
3	/				/	/
4				/	/	/
5	/				/	/
6	/				/	/
7	/				/	/
8		/			/	/
9		/				

Clearly some type conversions are required with parameter forms e and f. In the case of hexadecimal values it is assumed that the user knows the bit pattern to be passed and the only conversion to be applied is possibly a size change by zero filling or truncating on the left as necessary. In the case of nested procedures the result will be of known type and the normal type conversions will apply.

Further examples of commands are

OUTREAL <COS .5> 10 8

which will print the cosine of .5 in a field of size 10 to a precision of 8 decimal places, and

OUTREAL <SQRT<COS .5>> 10 8

which will print its square root.

In the case where a command parameter type is a pointer, the permissible substitutions are either ‘↑’ followed by a constant of the dereferenced type, a procedure which returns a pointer of the correct type, or a binary pointer coded in hexadecimal (somewhat unlikely).

Bounded pointers will normally require that a vector of constants be given enclosed in ‘[‘ ’]’. For example the CAPTION procedure requires a bounded pointer to a vector of LOGICAL8’s. An unlikely call on CAPTION may therefore be written as

CAPTION [%41 %42 %43 %44 %45]

In fact since vectors of LOGICAL8's very often represent character strings a special case is made of ADDR [LOGICAL8] which allows the brackets to be omitted and the actual character strings to be written as the parameter, provided that it does not contain spaces. Thus the above command would normally be written as

CAPTION ABCDE

Also a string can be enclosed in quotes. Thus a string containing spaces can be written as

CAPTION "THIS IS A STRING"

In the case of vectors of other types, the square bracket form must be used, and the elements of the vectors should be separated by spaces. For example, if a procedure SUM exists in a private library, which returns as a result the sum of a vector of integers, it might be used as follows

OUTI <SUM [10 20 51 64]>

2.3.5 Sets of File Names as Parameters

It is often convenient to be able to repeat a command which operates on files for each of a set of files which have similar names. The set of files to be operated upon is defined partly by the parameters of the command call and partly by a list of character strings on the current file. It is the special treatment applied to the '=' symbol in a character string parameter which generates the effect. This special treatment is that any command having one or more character string parameters which contain the = symbol will be automatically executed once for each character string in the current file. Consecutive strings on the current file will be substituted instead of the = symbols on each execution of the command. For example if the current file contains the five strings

011 012 013 025 035

then

1) DELETE SYS= is equivalent to

```
DELETE SYS011
DELETE SYS012
DELETE SYS013
DELETE SYS025
DELETE SYS035
```

2) COPYFILE USERX:LIB= USERY:LIB= is equivalent to

```
COPYFILE USERX:LIB011 USERY:LIB011
COPYFILE USERX:LIB012 USERY:LIB012
COPYFILE USERX:LIB013 USERY:LIB013
COPYFILE USERX:LIB025 USERY:LIB025
COPYFILE USERX:LIB035 USERY:LIB035
```

3) LISTFILE F= LPT* is equivalent to

```
LISTFILE F011 LPT*
LISTFILE F012 LPT*
LISTFILE F013 LPT*
LISTFILE F025 LPT*
LISTFILE F035 LPT*
```

A command FIND.FILES to assist in creating the list of filenames is described later in Section [5.3.1](#).

2.4 USING FILES AND DOCUMENTS

A high proportion of all job control commands, and many of their parameters, are concerned with setting up inputs and outputs. Physical input/output exists as “documents”, which may take a variety of forms: for example, a deck of cards; a lineprinter listing; a file, or a “conversation” at an interactive terminal. Within a job, documents are assigned to streams for processing, using commands which are described in detail in Chapter 3. However, many of the most commonly used system commands have parameters which relate directly to input and output documents, and they assign these documents automatically to streams. This kind of parameter is referred to as a “document name”. Formally the parameter type will be ADDR [LOGICALS] and the normal style in which an actual parameter is written is as a string of characters terminated by a space or newline.

2.4.1 File and Directory Names

Each user has his own set of files and their details will be kept in a directory associated with the user. The files will have names of up to 16 alpha-numeric characters. Subject to the necessary permissions being set, as described in Chapter 5, a user may access the files of another user. The notation for this is to precede the file name by “USERNAME:”, where USERNAME is the name of the root directory which the system provides for each user. Also a user may create a tree of files by introducing new subdirectories which are named like files. In this case a file is identified by preceding its name by the names of all the subdirectories on the path to the file. Thus in general a file name has the form

USERNAME:DIRECTORY NAME/DIRECTORY NAME/- - -/FILENAME.

This will subsequently be referred to as a *filename*. If the file is in a user’s own filestore the ‘USERNAME:’ can be omitted. Some commands only allow the latter form and this will be denoted by the term *local filename*.

At any place in the *filename* where there is a directory name it is also possible to have ‘.’ or ‘..’, to denote current directory or parent directory. In fact, the ‘.’ option has no effect on the overall *filename*, and is included only for completeness. The ‘..’ option takes the path back into the parent directory of the current directory.

There are some commands which allow directories (or subdirectories) to

be specified. The notation for these is as for files except the name will be truncated at the required directory. They will be referred to as *directory names* and *local directory names*.

When a subdirectory is selected, *local filenames* and *local directory names* will normally describe the path from the subdirectory. However, special forms of these names are allowed which start with ‘/’ and the path which follows this is assumed to be from the user’s root directory.

2.4.2 Document Names

Generally there are three main ‘kinds’ of document which may be used, and three corresponding forms of document name. These are:

- i) A file, in this or another user’s filestore, with the notation as described in Section 2.4.1.
- ii) An input/output device, or process. Output can be directed to any process in the system, by giving the process name followed by ‘*’ as the document name. This is mainly used to send output to device controllers, such as LPT* (the lineprinter) or PTP* (paper tape punch), for example LISTFILE FILE1 LPT*. Similarly, input can be solicited from a named input device, for example TABLET* for a graphics tablet. If output is to be discarded completely, the pseudo document name * may be used.
- iii) A stream which has already been assigned a document, using the DEFINE.INPUT or DEFINE.OUTPUT commands described in Chapter 3. In this case the special name STRn* (for stream n) may be used. Example LISTFILE FILE1 STR2*.

2.4.3 The Current File

There is one further kind of document, called the *current file*. This is like a file, but it exists only for the duration of the job unless some explicit action is taken to save it as a permanent file. The corresponding document name is zero, and so it can be used as a default parameter that is, if no document name is specified by a command, then the current file will normally be used instead.

At the start of a job the current file is undefined. Certain commands,

described later in this chapter, allow the user to set and alter the current file. Once it has been defined, omission of an input document name parameter, or the use of '0', will normally result in the current file being used instead. If this parameter form is used when no current file is defined, or when the current file has already been assigned to another stream, the currently selected input stream is used instead. Thus a PASCAL program in file "PROG1" may be compiled by the statement

PASCAL PROG1

whereas a PASCAL program on the current file is compiled by the command

PASCAL

If the above command is used when there is no current file the compiler will commence reading from the current (command) stream. This will result in the prompt '→' on the users terminal and a program may then be typed. It will be compiled line by line as it is typed. All three options result from the same internal action inside the compiler. It simply passes on the given parameter in a DEFINE.INPUT procedure call.

In the case of output documents, omission of the parameter causes the current file to be updated with whatever is output. There are a few commands which do not follow these general rules. For example, LISTFILE always defaults its output to the currently selected stream if the parameter is omitted.

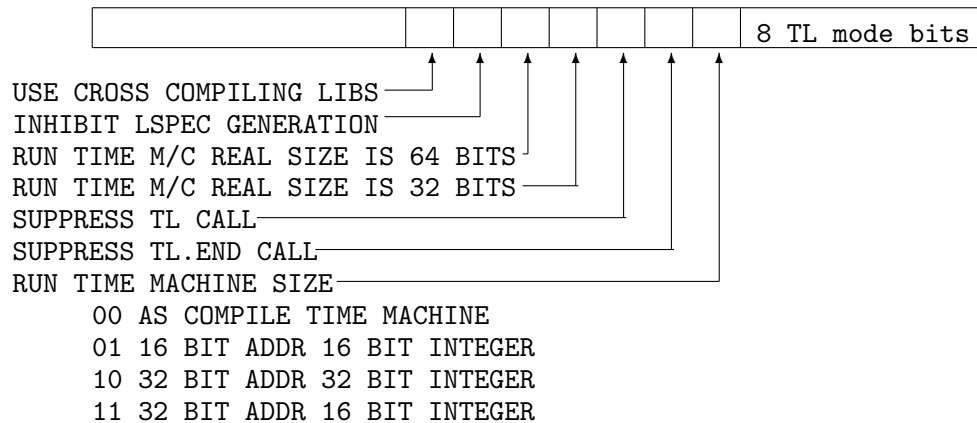
2.5 COMPILING AND RUNNING PROGRAMS

All the compilers operate through a common target language MUTL which is fully described in Chapter 18, although most users will only need to know about 'modules' (see Section 18.2.1), the TL mode bits (See Section 18.11, Item 6) and the diagnostic printing options (See Section 18.13). For details of specific languages the user should refer to the chapter of that name in Volume 2 of this Manual. However, the general pattern is as follows.

- 1) LANGUAGE (ADDR [LOGICAL8], ADDR [LOGICAL8], INTEGER, INTEGER)

The commands for the standard system compilers FORTRAN, BASIC, PASCAL, C, MUSL, are all of this form with the appropriate name substituted for LANGUAGE. P1 is the document name for the program to be compiled. P2 is the document name for the compiler output. Normally P2 will be the name of a file (or library) into which the binary program will be placed, or it may be 0 in which case the result the compilation becomes the current program. The current program is executable code, quite distinct from the current file. The remaining parameters specify mode and library information (see below), and may normally be omitted.

P3 has the bit significant encoding

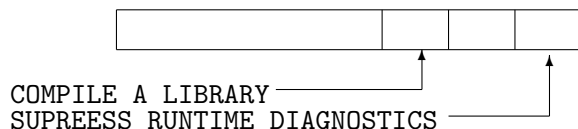


In the case of a single module program, the compiler will enclose the compiled code in the MUTL statements

```
TL
TL.MODULE
...
TL.END.MODULE
TL.END
```

It is only in situations where multi module programs/libraries are to be compiled that the TL and TL.END statements need to be suppressed.

The TL mode bits specify aspects of the compilation mode which once set hold for the complete compilation and are described in Chapter 18.11 Item 6, but the main ones are



Other aspects of mode, such as the control of optimisation and the positioning of fields within types are changed as required during a compilation by TLMODE, further details are in Chapter 18.

When compiling a library P4 specifies the maximum number of procedures permitted in the interface of the library. If P4 is zero a suitable system default is used.

Note that if a library is to be used from the command level it may only contain procedures whose parameters are of the types described in Section 2.3.4.

2) RUN (ADDR [LOGICAL8])

This command enters a program compiled by an earlier compilation command. P1 is the filename to which compiler output has previously been sent. The program specified also becomes the current program. This means that it may be subsequently re-run any number of times by calling RUN without having a file name.

3) LOAD.PROG (ADDR [LOGICAL8], INTEGER) INTEGER

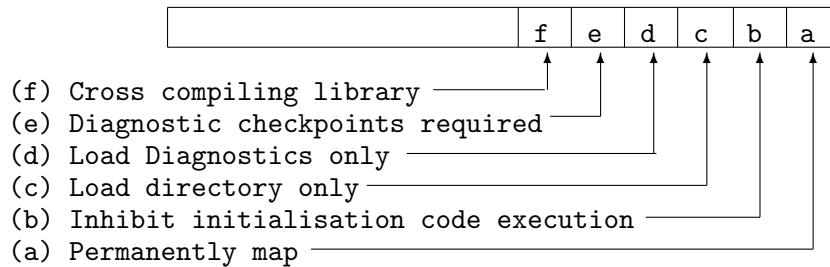
This command loads a program compiled by an earlier compilation command, but does not enter the program. The program may subsequently be run by calling RUN without a filename. Providing that P2 is zero, diagnostic checkpoints may be inserted in the code prior to running the program. Diagnostic checkpoints are described fully in Chapter 10. A zero result indicates that the program has been loaded successfully.

4) LIBRARY (ADDR [LOGICAL8], INTEGER)

This command loads the precompiled library specified by P1 and links it into the library directory structure so that its procedures become available both as commands and as procedures which may be called from subsequently compiled programs and libraries. P1 is the file name of a compiled library. P2

is the mode in which the library is to be opened. When a library procedure name is sought, the most recently loaded library directory is inspected first, therefore any user library procedures with the same names as system library procedures will take precedence over the corresponding system procedures. The facilities provided for creating library files are described more fully in Chapter 7.

P2 has a bit significant encoding and is interpreted as follows:



Notes

- (a) A zero value indicates that the library is to be used in the ‘normal’ mode for the machine in question. Normal mode on machines with large virtual memories means that the library document is opened into the virtual memory. For machines with small virtual memories the norm is to overlay (MAP) the library document into virtual memory on each call, a value of one indicates that the library is to be permanently mapped.
- (b) When set it indicates that the initialisation code of the library is not to be executed.
- (c) When set it indicates that only the directory of the library is to be loaded.
- (d) When set it indicates that the diagnostics of the library are required. This option enables diagnostics to be accessed for system libraries.
- (e) When set it indicates that diagnostic checkpoints may be inserted in the code of the library. Diagnostic checkpoint facilities are described fully in Chapter 10.

- (f) When set it indicates that this library will be inspected only by compilers where the use of cross compiling libraries bit is set.

5) RELEASE.LIB (ADDR [LOGICAL8], LOGICAL)

This command unloads a user library (P1) and unlinks it from the library directory structure. P1 should be the final actual filename rather than a full *filename* specifying the access path.

If $P1 = 0$ all libraries except the basic system library are unloaded.

2.5.1 Commands for job sequencing, etc.

1) STOP (INTEGER)

This command terminates the job, after ending its output streams and returning any online terminal to its *primary* connection state. The current file is discarded. P1 is a “reason code” for the termination. Zero indicates normal termination. A negative value indicates an abnormal (error) termination, in which case any remaining file outputs are not updated. (see Chapter 3.2.5 DEFINE OUTPUT.)

2) IN (ADDR [LOGICAL8], INTEGER)

This command causes job control commands to be read from the specified input document (P1). If P1 contains the string “-1” the previously selected input document is restored. P2 specifies the maximum number of command errors expected, when this count is exceeded the previously selected input document is restored.

3) RUN.JOB (ADDR [LOGICAL8], ADDR [LOGICAL8], ADDR [LOGICAL8])

This command is used within one job to initiate execution of another as a separate (background) job. P1 is the name of an input document on which job control commands for the new job may be found. If it is left unspecified and no current file is defined, the commands for the new job are input from the current stream terminated by ‘/’ at the start of a line. P2 is the name of the supervisor to which the job is to be submitted, in the form “process name*” or “process name*/machine name”. If P2 is left unspecified, JOB is assumed. P3 is the “header” for the new job.

If P3 is left unspecified, a header line of the form “username password title” will be generated, where username and password specify the current user and title is a unique jobname.

4) KILL (LOGICAL64, INTEGER)

This command may be called by a user to terminate the specified process P1. It will only operate if the process is running under the same username as the current user (or if the current user is privileged). The parameter P2 states a reason for killing the process.

2.6 A JOB CONTROL EXAMPLE

In the example given below the computer output is distinguished from the user’s input, by the former being bold and the latter being in italics. On the actual system the fonts will normally be the same.

The first command used after the log-in line is NEW, which is used to input to the current file a Fortran program for computing prime numbers. This is followed by the FORTRAN command which compiles the program but finds one error. This is corrected by editing the current file. The first edit statement copies to line 16 and 'windows' the line. The second means

```
delete IR
insert 'RI'
and window.
```

Positions in the file may also be selected by context but it is more convenient to use line numbers when a compiler gives them with the error reports. At the second attempt the program compiles correctly and it is entered by the RUN command. Since the program contains a call for the READ procedure which reads an integer, it prompts for data. When it is given the integer 20 it computes all prime numbers less than 20, and returns to command mode as a result of executing the STOP. However, a mistake in the program causes all ones to be printed. On examining the program it is clear that PRIMES(P) should have been set to I not 1 so this is corrected by editing. The program is recompiled and then runs correctly. At this point the current file is saved and listed, and the user logs out.

M JOB

Type job information

Username Password Jobname (Options)

D4 XXX DEMON

DEMON 10:20:49 09-APR-84

**NEW

```

/      INTEGER SIEVE(1000),PRIMES(200),LIM,P,J
/      WRITE(*,200)
/      200 FORMAT(1X, 'TYPE PRIMES LIMIT,I3 FORMAT PLEASE')
/      READ(*,100)LIM
/      100 FORMAT (I3)
/      DO 1 I =1,LIM
/      1  SIEVE(I) = 0
/      P = 0
/      DO 2 I = 2,LIM
/      IF (SIEVE(I) .NE. 0) GO TO 2
/      P = P + 1
/      PRIMES(P) = 1
/      DO 3 J = I,LIM ,I
/      3  SIEVE(J) = 1
/      2  CONTINUE
/      WIRTE( *,201)LIM
/      201 FORMAT(1X, 'PRIMES UP TO ',I3)
/      WRITE(*,202) (PRIMES(I), I = 1,P)
/      202 FORMAT(/(1X,8(I3,4X)))
/      STOP
/      END
/      *END
//

```

**FORTRAN

SEGMENT 3 CREATED

SEGMENT 4 CREATED

MUSS.12 FORTRAN APR.84 AT 10:27:47 ON 09-APR-84

1.16 WIRTE(*,201)LIM

↑

ERROR invalid syntax

1 **ERROR** MESSAGE(S) ISSUED DURING COMPILATION

FINISHED AT 10:27:57 ON 09-APR-84

```

**EDIT
→C+15W
1.16
~↑~          WIRTE(*.201)LIM
→D/IR/1/RI/
→W
1.16
          WRI~↑~TE(*,201)LIM
→E
<CFILE> AT 10:32:57 ON 09-APR-84 OK
**FORTRAN
SEGMENT 5 CREATED
SEGMENT 6 CREATED

MUSS.12 FORTRAN APR.84 AT 10:33:30 ON 09-APR-84
FINISHED AT 10:33:31 ON 09-APR-84.
**RUN
TYPE PRIMES LIMIT,I3 FORMAT PLEASE
→020

PRIMES UP TO 20

      1      1      1      1      1      1      1      1
STOPPED AT 10:34:48 ON 09-APR-84 LINE 1.20
**EDIT
→C/PRIMES(P)/W
1.12
~↑~          PRIMES(P) = 1
→D/1/I/I/W
1.12
          PRIMES(P) = I~↑~
→E
<CFILE> AT 10:36:26 ON 09-APR-84 OK
**FORTRAN
SEGMENT 7 CREATED
SEGMENT 8 CREATED

MUSS.12 FORTRAN APR.84 AT 10:36:53 ON 09-APR-84
FINISHED AT 10.37.52 ON 09-APR-84
**RUN

```

```
TYPE PRIMES LIMIT,13 FORMAT PLEASE
```

```
->020
```

```
PRIMES UP TO 20
```

```
    2    3    5    7   11   13   17   19
```

```
STOPPED AT 10:37:20 ON 09-APR-84 LINE  1.20
```

```
**SAVE DEMO
```

```
**LISTFILE DEMO
```

```
      INTEGER SIEVE(1000),PRIMES(200),LIM,P,J
```

```
      WRITE(*,200)
```

```
200  FORMAT(1X,'TYPE PRIMES LIMIT,I3 FORMAT PLEASE')
```

```
      READ(*,100)LIM
```

```
100  FORMAT (I3)
```

```
      DO 1 I =1,LIM
```

```
1    SIEVE(I) = 0
```

```
      P = 0
```

```
      DO 2 I = 2,LIM
```

```
        IF (SIEVE(I) .NE. 0) GO TO 2
```

```
        P = P + 1
```

```
        PRIMES(P) = I
```

```
        DO 3 J = I,LIM ,I
```

```
3    SIEVE(J) = 1
```

```
2    CONTINUE
```

```
      WRITE(*,201)LIM
```

```
201  FORMAT(1X, 'PRIMES UP TO ',I3)
```

```
      WRITE(*,202) (PRIMES(I), I = 1,P)
```

```
202  FORMAT(/(1X,8(I3,4X)))
```

```
      STOP
```

```
      END
```

```
      *END
```

```
**STOP
```

```
10:38:25 09-APR-84 CPU USED 14 STOP REASON 0
```


MUSS

USER MANUAL

VOLUME 1

CHAPTER 3

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

3 I/O STREAM MANAGEMENT

3.1 INTRODUCTION

The input/output system was briefly introduced in Chapter 2. It was explained that physical Input/output exists as ‘documents’, which may take a variety of forms: for example, a deck of cards; a lineprinter listing; a file, or a ‘conversation’ at an interactive terminal. The input/output facilities described in this Chapter and Chapter 4 provide programs with a common interface to all kinds of input/output documents, so that the programmer need not be aware of the actual source or destination of a particular document. They also provide a common interface between the input/output facilities of all of the programming languages, so that a file generated by a FORTRAN program may be read by a BASIC program, and vice versa.

High-level language programs perform input and output operations using the facilities of the language in which they are written. These are described in the relevant programming language manuals. However, all of the programming language facilities in MUSS are implemented in terms of the basic operations described in the next Chapter and some languages use the basic operations directly. The exact relation of the programming language facilities to the basic input/output system is described separately for each language in Volume 2 of this Manual.

Input and output operations in MUSS are organised in terms of logical input/output streams. Each process may have several input and output streams in use at any one time, and may switch between these at will. There may be some variation in the number of available streams in different implementations of MUSS but the norm is 32 of each. The most important functions of the basic input/output system are to provide the means for

- i) Assigning actual documents to input and output streams (usually by job control commands).
- ii) Selecting the particular stream to be used for each input/output operation.
- iii) Performing actual input/output operations on a stream.

The basic operations provided are used by particular compilers to build up the complete set of input/output facilities for the corresponding programming languages. This Chapter describes the stream assignment operations; the others are described in Chapter 4.

3.2 CHARACTERISTICS OF INPUT AND OUTPUT STREAMS

3.2.1 Structure of an Input/Output Stream

The information in an input/output stream is structured as a series of records. In its most simple form, where a document is just a sequence of characters, this structure simplifies to a single record. Each record is preceded by a 32-bit count indicating the size of the record. For compatibility across machines, this count is unpacked, with the least significant byte first. A count of -1 indicates the last record of a document.

Access to a stream is normally sequential, although the basic system supports facilities for positioning a stream at a particular point within a record for processing the information sequentially from that point. Thus random access can be implemented by forming an external index, perhaps in another stream.

3.2.2 Sectioned Streams

The actual document assigned to a stream may be either a file or a message (or sequence of messages). Each message, if there are more than one, is referred to as a *section* of the stream. The division into sections is achieved by explicit calls of the Break Input and Break Output commands.

3.2.3 Message Streams – Messages and Synchronisation

A message stream consists of a sequence of messages from or to another process. Streams which communicate with peripheral devices are of this type. Since the devices are controlled by special processes. Thus for example input and output on an interactive terminal uses message streams.

The system distinguishes between two types of message, they are called *long* and *short* messages. Long messages are used for communicating with bulk input/output devices (such as card readers lineprinters), and short messages for interactive devices. The user need not be aware of the distinction in the case of input streams, as the system will automatically read from whichever

kind of message is received. For output, however, the user must define which kind of message is to be used on any message streams created. If short messages are specified, the section size is determined by the size of the buffer used – about 100 characters – and multiple sections will be generated automatically.

In the case of message input streams, the user also has to define what action the system should take when an attempt is made to read beyond the last available message. There are two options: to wait until further messages arrive, or to signal a fault. The former is suitable for input from an interactive terminal, and such a stream is referred to as an *online* input stream. The latter is more appropriate for batch input, which is normally all present before the job is started; such a stream is called an *offline* input stream.

In the case of input from a named device, it is also possible to control the synchronisation between the device and the process. With some devices, such as a graphics tablet, it is desirable to have the device reading data only when data is specifically required by the user program. In this case, data is solicited from the named device at the time of breaking the input stream.

For message output streams, the user may also wish to indicate that the output is to be synchronised to the operations of the receiving process. In interactive operation, for example, it is undesirable for a large amount of output to be generated by a job in advance of it being printed. Two means of synchronisation are possible, one automatic and the other explicit. In the automatic case, the job is suspended when the message is sent, and can only be freed by the receiving process. This is used to synchronise with an output device, as the output device controllers will perform the freeing operation, if necessary, on completing the output of a document. The alternative method is to request the receiving process to send a message in reply; this message must then be detected and read explicitly. This option enables a job to request notification when output is performed, without actually being halted.

3.2.4 IO Streams

An IO stream is a stream on which both input and output operations may be performed. A *single pointer* is used for both input and output. Thus,

interleaved sequential input and output operations will operate on consecutive elements of the stream. Usually the strictly sequential access will be broken by the user repositioning the pointer.

At any instant, a record within an IO stream is notionally set up for either input or output, although both types of operations will successfully operate on the stream. A change from one mode to the other is performed by calling SET.I.REC or SET.O.REC as appropriate. The distinction arises due to the assumed bound of the current record, which is set to the previously defined size of the record in the input case, and the size of the document in the output case. This distinction does not affect normal IO operations, but only the behaviour of the stream on reaching the end of the record. For example, if a previously defined record is to be extended, a call to SET.O.REC is required to allow a new, larger record to be written. If trapping is required on reaching the end of a previously defined record, then SET.I.REC should be called to delimit the input and output operations. An IO stream is normally initialised for output, except in the case of files where the append access mode is specified (see Section 3.2.5 note g)).

An IO stream occupies both an input and an output stream. Releasing or re-defining either the input or the output associated with an IO stream causes both to be discarded.

3.2.5 Stream Modes

There are four commands which provide for the creation of streams

- 1) DEFINE.INPUT
- 2) DEFINE.OUTPUT
- 3) DEFINE.IO
- 4) DEFINE.STRING.IO

Each of these has a mode parameter which allows options to be selected as shown below. Not all options apply to all commands, and this is shown below by 'n' in the appropriate position of the tables headed Input, Output and IO. There is no column allocated to the 4th command, the string IO, which is not normally used directly by the user. In this case, the only options relevant to it are the last three.

Notes:

- a) If this bit is set and the input is obtained a character at a time using IN.CH or other procedures which themselves use IN.CH, then the end of record character (RS = %1E) will be generated after the last actual character of each record.
- b) The norm is that input composed of several records may be read as an unstructured character string by means of IN.CH. However, if this bit is set and the input is obtained a character at a time using IN.CH, then a trap will be entered if an attempt is made to read from one record to the next.

- c) If this is set and the input is read using IN.CH an attempt to read beyond the last character in a section will generate a 'section separator' character (FS = %1C).
- d) Normally a message stream containing several messages may be read as continuous uninterrupted input (providing no attempt is made to reposition the input pointer in a different section). If this bit is set any attempt to read across a section boundary will result in a trap entry.
- e) When a Nil or Zero document name is given the default is normally to the current file, and in the case of input to the current stream if there is no current file. This bit causes output to default to the current stream. and in the case of input it prevents the second default option (current stream) from being used. Thus, a trap will be entered if there is no current file available when an input is defined as the nil filename and this bit is set.
- f) This only has effect when the input source is a file. If the bit is set a copy of the file is made at the time of defining the stream, otherwise the file is used directly. Changes made to a file in this way are only guaranteed to be written back to disc when a SECURE.SEGMENT command is used. The BREAK.OUTPUT and END.OUTPUT commands will do this automatically.
- g) Indicates the action to be taken when the file mentioned in DEFINEIO is an existing file. The norm is for the file to be opened and the pointer to be placed at the first record. Several ways exist in which the pointer may be moved including WIND to position at the end of the file for appending purposes. Intermediate records may also be changed although normally only to other records of the same size. Also any record position at which the pointer has been placed can be declared to be the new end of the file by 'END.POS'. It may subsequently be appended to from this position but attempts to read beyond this position will be faulted.
- h) This bit is only applicable to message input streams, and determines the behaviour when the end of a section has been reached and a new message has to be read. If there are no messages on the message channel, the default action is to trap the process when it attempts to read from the stream. If h is set, messages are assumed to be coming

from an online source, such as a terminal, and the action is to halt the process until a new message arrives.

- i) In the case of input this indicates that the input should always be from the header, even for a long message – see Section 3.2.4. In the output case, output is in short messages.
- j, k) Normally when a process ends, its output streams are broken and the accumulated output is sent to its destination. If a process is trapped the same thing normally happens to its message output but new files will not be created since this could destroy existing files of the same name. However, this action can be overridden by bits j and k. If bit j is set the file output will be filed even if a program traps. If bit k is set no output messages will be dispatched if a process traps.
- l) Section 3.2.3 gives further information on the synchronisation and conversational features of the message system. In the case of an output or IO stream, this field determines the behaviour of the current process on sending a message. For an input stream, this information determines the behaviour of a connected device when sending data to the process.
- m) Is a 4-bit field which allows any channel of the destination process to be specified as the destination of message output.
- n) This bit should be set if it is anticipated that accesses to the document will be at random positions, rather than sequential accesses. This enables the input/output system to optimise, wherever possible, the buffering of the document.

3.3 ASSIGNING DOCUMENTS TO STREAMS

The input and output streams of a process are identified to the basic I/O system by stream numbers. Input and output stream 0 are normally used for the “default” input and output stream associated with a job. For example a job submitted on cards will automatically have input stream 0 assigned to the card input document, and output 0 to the lineprinter associated with the input station; for an interactive job, input and output streams 0 are automatically assigned to the terminal input and output respectively. The remaining streams are assigned by the user to any other documents which are needed during the course of the job.

Documents may be assigned to streams either explicitly by the user, or automatically by the software he is using. In the case of standard system software such as compilers and editors, and of most applications packages, the assignment is automatic – the user supplies a “document name” as a parameter, and the corresponding document is assigned behind the scenes to a stream. The main use for explicit assignments arises in running a user’s own programs. Here, the language implementor will specify the correspondence between the stream numbers of MUSS and the “logical channels”, “files”, or whatever are operated on by the program. It is then up to the user to assign documents to the appropriate streams prior to entering his program. Sometimes it is desirable to operate on an explicitly assigned stream using system software, and special document names are available to allow this to be done.

3.3.1 Procedures for Defining and Releasing Streams

1) INIT.IO ()

This procedure completely re-initialises all input and output streams, abandoning any which are currently defined. It is performed automatically at the start of a job and is not normally used again, but it can be used as a drastic recovery measure in the case of very serious errors.

2) DEFINE.INPUT (INTEGER, ADDR [LOGICAL8], INTEGER32) / INTEGER

This procedure defines a new input stream, overriding any existing definition of this input stream. If the stream specified is defined already, any input being processed (i.e. the current section) on the stream is discarded. If the currently selected input stream is re-defined, it automatically remains selected but with the new document assigned to it.

P1 specifies the stream number to be defined and is normally 0-31 or -1. If P1 is negative a free stream number will be selected by the system. The stream number actually used is always returned as the integer result of the procedure.

P2 specifies the document which is to be assigned to the stream, and may take any of the forms

- (a) A filename, indicating that the document is a file. If the filename contains the character ‘:’, it will be interpreted as username:filename, and the file belonging to the user named will be accessed provided that permission has been given. If no username is specified, the currently selected directory is used.
- (b) Zero, indicating that the current file is to be used. The current file may only be assigned to one input stream at a time. If no current file is available, the currently selected input stream will be assigned instead provided that the mode (P3) allows this and that P1 was negative.
- (c) A message channel CHO* CH1* CH2* CH3* CH4* CH5* CH6* CH7*, indicating that the document will consist of messages addressed to the specified message channel.
- (d) A stream name – STR0*, STR1*, etc. This enables an already existing stream (already set up by a preceding DEFINE.INPUT) to be specified as a parameter. If P1 is negative, the stream number of the named stream is returned; otherwise the document on the stream named is re-assigned to stream P1 and the named stream becomes undefined.
- (e) A named device followed by a ‘*’, for example, TABLET*. The device is connected to the current process and a message channel is assigned to receive messages from the specified device. Thereafter, the stream behaves as if it were a message stream, as in (c).
- (f) A document within a module. The name of the document is specified as ‘module name>document name’, where the module name has the same structure as the filenames described in (a).

P3 gives the mode to be associated with the stream, and is interpreted as indicated in [3.2.5](#).

- 3) DEFINE.OUTPUT (INTEGER, ADDR [LOGICAL8], INTEGER32, INTEGER32) / INTEGER

This procedure defines a new output stream, overriding any existing definition of this output stream. If the stream is defined already, any

output produced and not yet dispatched (i.e., the current section) is discarded. If the currently-selected output stream is re-defined, it automatically remains selected but with the new document assigned to it.

P1 specifies the stream number to be defined. If P1 is negative a free stream number will be selected by the system. The stream number actually used is always returned as the integer result of the procedure.

P2 specifies the document which is to be assigned to the stream, and may take any of the forms

- (a) A filename, indicating that the document is to become a file. If the filename already exists, the existing copy will be overwritten when the stream is broken. If the filename contains the character ‘:’, it will be interpreted as username:filename and the file belonging to the user named will be updated provided that permission has been given. If no username is specified, the currently selected directory is used.
- (b) Zero, indicating that the document, on completion, is to become the current file or output to the currently selected stream, depending upon the mode setting. The “current stream” option is only valid for negative P1.
- (c) A process name followed by a ‘*’, indicating that sections of the document are to be sent as messages to the specified process. This option is used for output to standard system devices – for example LPT* for a lineprinter, PTP* for a paper tape punch. If the process name contains the character ‘:’, it will be interpreted as machine name:process name, and the output will be sent to the machine named. Otherwise, the current machine is assumed.
- (d) A stream name – STRO*, STR1*, etc. This enables an existing stream (already set up in an earlier DEFINE.OUTPUT) to be specified as a parameter. If P1 is negative, the stream number of the named stream is returned; otherwise the document on the stream named is re-assigned to stream P1 and the named stream becomes undefined.

- (e) A reply name – REPO*, REP1*, etc. This indicates that any output produced is to be sent as a reply to the last section received on the specified input stream.
- (f) The character ‘*’ alone, meaning that any output produced on the stream is to be discarded.
- (g) A document within a module. The name of the document is specified as ‘module name>document name’ where the module name has the same structure as filenames described in (a). If the specified module already contains a document of that name, it will be overwritten when the stream is broken. In other cases, a new entry will be made in the module directory.

P3 gives the mode to be associated with this stream, and is interpreted as indicated in [3.2.5](#).

P4 gives the maximum section size in bytes of each section of output. For short message streams this parameter is not used. A zero gives an installation dependent default which is normally the hardware segment size. If a process exceeds the stated size of any individual section it will be trapped.

4) DEFINE.IO (INTEGER, ADDR [LOGICAL8], INTEGER32, INTEGER32) / INTEGER

This procedure is used to define a stream which may be used both as input and output. If the specified stream exists already (either as an input stream, or an output stream, or both) the existing definition is overridden and any existing input/output is discarded.

P1 specifies the stream number to be defined. If P1 is negative, a free stream (i.e. a stream number for which neither input nor output is defined) will be selected by the system. The actual stream number used is always returned as the integer result of the procedure.

P2 specifies the output destination. This may take any of the forms allowed for DEFINE.OUTPUT. If the stream name form STRO*, STR1*, etc., is used, the stream must already be an IO stream. The stream number and P2 must specify the same stream name. The default

to the current stream is not allowed. When the specified destination is the name of an existing file, and the mode specifies append access, the stream will be initialised to contain this file. Otherwise a scratch segment will be allocated, which will normally be dispatched to the specified destination when the stream is closed. In addition to the above options this parameter may take a form which is particular to `DEFINE.IO`. This is a name followed by “↑” e.g. “`PROC↑`”. The name `PROC1` must be that of a library procedure of a currently open library having one parameter of type integer. The effect is that the specified procedure will be called at break output time with the stream number passed as the parameter. Thus the specified procedure intercepts all output on the stream.

Prior to dispatch a scratch segment will have its size changed down to the minimum which will accommodate all its records. However, there will be no size change to a file accessed in append mode.

P3 specifies the mode for the stream, and is interpreted as indicated in [3.2.5](#).

P4 is the document size, as in `DEFINE.OUTPUT`.

5) `CHANGE.DEST (INTEGER, ADDR [INTEGER])`

This procedure enables the destination process for an existing message output stream to be changed, without otherwise redefining the stream in any way. P1 gives the stream number whose destination is to be changed. P2 is a vector containing the system process number (SPN) process identifier (PID), channel number for the new destination and a sequence number. This information may be found by calling `I.SOURCE` if the output is to reply to some input already received, or `LOOK.UP.PROCESS` if the output is to an explicitly named process.

6) `DEFINE.STRING.IO (INTEGER, ADDR [LOGICAL8], INTEGER32) / INTEGER`

This procedure enables a vector of bytes specified by P2 to be assigned to an IO stream P1, so that subsequent input and output operations on this stream access the byte vector. If P1 is negative a free stream number will be selected by the system. The stream number actually

used is always returned as the integer result of the procedure. P3 gives the mode, as indicated in 3.2.5.

The document produced within the byte vector may comprise a number of records, and hence will contain record housekeeping information as described in 3.2.1. The stream is initially set up for writing a record up to the size of the byte vector, allowing for a four byte record marker at the beginning and end of the vector. Vectors computed using other than the standard input and output operations should allow for these record markers when defining and manipulating the vector.

7) END.INPUT (INTEGER, INTEGER)

This procedure is used to release an input stream which is no longer required. P1 specifies the stream to be released.

Normally, programs which define input streams release them using END.INPUT with P2 positive. This will release the stream, unless it was last assigned using the STRn, or “current stream” form of document name (see DEFINE.INPUT), in which case the stream continues to exist. If P2 is negative, the stream will be released regardless. (Negative P2 is used when the stream is being released abnormally, as a result of an error.)

8) END.OUTPUT (INTEGER, INTEGER)

This procedure is used to release an output stream or an I0 stream which is no longer required. P1 specifies the stream to be released

If P2 is positive, the final section of the output stream is dispatched to its destination in the usual way, before being released. For negative P2, the action depends upon the mode associated with the stream, and the type of stream. For a message output stream, the output will be dispatched unless the DISCARD bit (k) in the mode (see Section 3.2.5) is set. For other stream types, the output will be discarded unless the SAVE bit (j) in the mode as set. (Negative P2 is used when the stream is being released abnormally as a result of an error. By default, message streams are dispatched in this case, while file streams are not. The SAVE and DISCARD bits allow this default action to be overridden if necessary.)

As with END.INPUT, the stream is not released for positive P2 if it was last assigned using the STRO*, or “current stream” forms of document name.

9) PROCESS.NAME (ADDR [LOGICAL8]) / ADDR [LOGICAL8]

This procedure scans the document name P1 to identify the type of document corresponding to it. PWW1 is set to the document type, as follows:

0	Current file (or current stream)
1	File
2	Process
3	Reply stream (REPn*)
4	Stream number (STRn*)
5	Scratch output document (*)
6	Message input document (CHn*)
7	Procedure (PROC↑)
8	Module document (Module>doc)

Additional information is returned according to the type of document

File	– PWW1 contains the directory name. The result of the procedure is a pointer to the filename (pathname).
Process	– PWW1 contains the process name component and PWW2 contains the machine name.
Reply or stream	– PWW2 holds the stream number, i.e. the ‘n’ component of REPn*, STRn*, or CHn*.
Procedure	– The result of the command is a pointer to the procedure name.
Module document	– PWW1 contains the directory name. The result of the procedure is a pointer to the full module>document name.

Control characters appearing in the document name delimit the name so that, for example, leading or trailing spaces or nulls are discarded.

Examples of Use of Stream Definition Commands

Of the commands described in this section, only DEFINE.INPUT and DEFINE.OUTPUT are normally used as job control commands. INIT.IO cannot sensibly be used as it leaves all streams undefined, and so there is nowhere from which further commands could be read. The following examples show the use of DEFINE.INPUT and DEFINE.OUTPUT in some common situations.

- | | |
|---------------------|---|
| i) DI 0 CHO* | This is the command used automatically at the start of a batch job, to initialise its default input stream. A message input stream is defined, with all terminators suppressed. Any attempt to read a new section with no message present results in a trap. |
| ii) DI 0 CHO* %80 | This is used automatically at the start of an interactive job to initialise its default input stream. The difference between this and i) is that in this case, reading a new section with no message present results in a prompt being output on the selected prompt stream and the process being halted until a message arrives. |
| iii) DI 1 MFY | This assigns the file MFY to stream 1 This is the kind of command normally used to set up input streams prior to entering a user program. |
| iv) DI 5 | This is similar to case iii) above, but the current file is assigned to stream 5. |
| v) DO 0 LPT* | This is the command used automatically at the start of a batch job to initialise its default output stream. A long message stream, with an installation-defined size limit is defined to be directed at the lineprinter control process. |
| vi) DO 0 REPO* %180 | This is the command used automatically at the start of an interactive job, to initialise its default output stream. A short message stream |

is defined, synchronised with the output device so that the process awaits printing of each output action before continuing. Each section of output is sent as a reply to the process from which the current section of input stream 0 was received (i.e. the interactive terminal).

vii) DO 1 OFILE

This assigns the file OFILE to output stream 1 – i.e. on a ‘BREAK.OUTPUT’, the stream will be filed as OFILE. Only one section will be permitted, and this will be of an installation-defined length. item[viii) DO 5 LPT* 0 10000] This defines an output stream, directed at the lineprinter, consisting of sections of at most 10 thousand bytes.

Examples of Automatic Stream Definition

Many of the examples given in Chapter 2 were in fact special cases of this facility.

i) NEW FILEX

FILEX is an automatically assigned output stream, directed at the file specified.

ii) FORTRAN FILEX

An automatically assigned input stream, causes the source program on FILEX to be compiled.

iii) EDIT FILEX FILEY

Automatically assigned input and output streams, FILEX is edited to produce FILEY.

iv) FORTRAN

This causes the source program on the current file to be compiled if one exists, otherwise the source program is taken from the currently selected input stream.

v) EDIT

The current file (if one exists) is edited to produce a new current file. In this case, if no current file exists, a fault is generated.

vi) LF FILEX LPT*

The input stream is from the FILEX, the output stream is set up, with default parameters,

to the lineprinter. item[vii) LF FILEX STR1*]
In this case, the output stream is defined to be stream 1, which must have been set up previously by a DEFINE.OUTPUT command. This is useful (a) if it is required to use an output stream with parameters other than the defaults, and (b) if it is required to append the listing to the end of a stream which already has some output on it. item[viii) LF] The current file is listed on the currently-selected output stream (c.f. v) above). A fault is generated if no current file exists.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 4

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

4 BASIC INPUT/OUTPUT OPERATIONS

The procedures in this Chapter are intended mainly for use in programs, rather than as job control commands, and they enable a program to perform the following functions

- i) Select a particular stream to be used for subsequent input or output operations until another stream is selected, and to enquire about the currently selected streams.
- ii) Explicitly break an output stream into sections.
- iii) Discover the mode, and other information, associated with a stream.
- iv) Perform actual input/output operations.

The user should be aware that all I/O documents in MUSS consist of one or more records so that the notion of position, in general, is two dimensional, namely

record number, and
position within record.

When it is known that a file contains only one record, which may in fact be an arbitrary string of characters containing many lines separated by newline characters, the record positioning commands can be ignored.

4.1 PROCEDURES FOR STREAM SELECTION, ENQUIRY ETC

1) SELECT.INPUT (INTEGER)

This procedure selects input stream P1 as the current input stream. All subsequent input operations until the next call of SELECT.INPUT will read from this stream. A fault is indicated if the stream is not defined at the time of selection.

On selecting a new input stream, the current position in the currently selected stream is remembered, so that upon re-selection input can resume from the present position.

2) SELECT.HEADER ()

In the case of an input section which is a long message, this procedure causes subsequent input on this section to be from the header. For short messages, input re-commences from the start of the section. On switching to a new section, the header or main document is selected according to the mode of the input stream (see Section 3.2.5).

3) SELECT.DOC ()

This procedure selects the main document of the current input stream. For a short message, the action is identical to SELECT.HEADER. On switching to a new section, the header or main document is selected according to the mode of the input stream (see Section 3.2.5).

4) BREAK.INPUT (INTEGER)

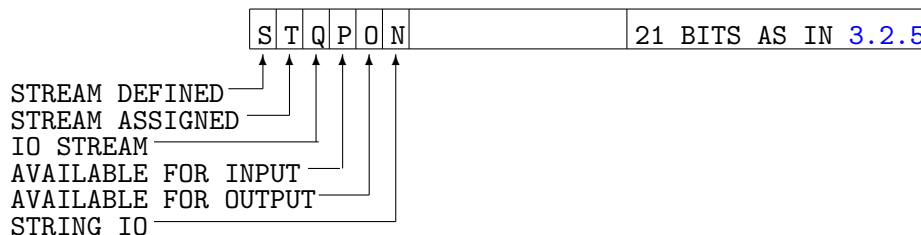
This procedure advances input stream P1 explicitly to the next section. If P1 is negative, the current stream is advanced.

5) CURRENT.INPUT () / INTEGER

This procedure returns as its result the stream number of the currently selected input stream.

6) I.MODE () / INTEGER32

This procedure returns the mode bits for the current input stream, as described in Section 3.2.5, with the most significant six bits encoded as follows



7) I.SEG () / INTEGER

This procedure returns the segment number (if any) for the current input

stream. A negative result implies there is no segment – i.e. the current section is a short message.

8) I.REC () / INTEGER32

This procedure returns a 32-bit index into the current input stream of the start of the current record. This may be used subsequently as a parameter to SET.I.REC and SET.O.REC.

9) I.POS () / INTEGER32

This procedure returns a 32-bit index to the current character of the current input record. This may be used subsequently as a parameter to SET.I.POS and SET.O.POS.

10) I.SIZE () / INTEGER32

This procedure returns, as a 32-bit integer, the total size in bytes of the document on the current input stream.

11) R.SIZE () / INTEGER32

This procedure normally returns, as a 32-bit integer, the number of bytes in the current record. If the stream is an IO stream, which is selected for output, the number returned will be the maximum size of the current output record.

12) I.SOURCE (ADDR [INTEGER])

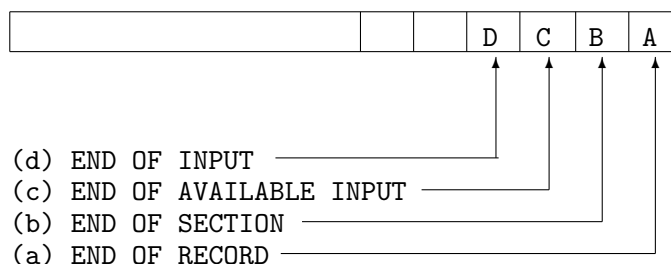
This procedure yields the identification of the process which sent the current section of the current input stream, in a vector of four integer elements supplied as a parameter. A trap is entered if the input stream is not a message stream.

```
P1[0] := system process number (SPN)
P1[1] := process identifier (PID)
P1[2] := channel to which replies are to be sent
P1[3] := message sequence number.
```

This information is in a suitable form for use as a parameter to CHANGE.DEST or SEND.MESSAGE.

13) I.ENQ () / INTEGER

This procedure allows a process to enquire whether any input is currently available at the current input stream. The integer result is interpreted as follows:



Notes:

- (a) is set when the last byte in the current record has been read and the end of record character has been returned (if required).
- (b) is set when the last byte of a section has been read and the end of file character has been returned (if required).
- (c) is set then the end of a section has been reached and no further sections are available. For an online message stream (see Section 3.2.3), subsequent attempts to input an item sequentially from the stream will cause the process to be halted. In all other cases, attempts to perform further input will be faulted.
- (d) is set when the last byte of a stream has been read. A subsequent attempt to input an item sequentially from the stream will generate a fault.

14) I.DOC () / ADDR [LOGICAL8]

This procedure returns the name and type of the current input document. Additional information about the document type is also returned in PW1 and PWW1. PW1 is set to the type of document and determines the interpretation of the procedure result and PWW1, as follows:

	PW1	Procedure result	PWW1
Current file	0	Nil	0
File	1	Pointer to filename	Directory name
Process	2	Nil	Process name
Message channel	6	Nil	CHn
Module document	8	Pointer to module>document name.	

15) SELECT.OUTPUT (INTEGER)

This procedure selects output stream P1 as the current output stream. All subsequent operations until the next call of SELECT.OUTPUT will output to this stream. A fault is indicated if the stream is not defined at the time of selection.

On selecting a new output stream, the current position in the currently selected stream is remembered, so that upon re-selection output can resume from the present position.

16) CURRENT.OUTPUT () / INTEGER

This procedure returns as its result the stream number of the currently selected output stream.

17) O.MODE () / INTEGER32

This procedure returns the mode bits for the stream encoded as in I.MODE.

18) O.SEG () / INTEGER

This procedure returns the segment number associated with the current output stream. A negative result implies that there is no segment – i.e. the stream is a short message or string stream.

19) O.REC () / INTEGER32

This procedure returns a 32-bit index to the current record of the current output stream. It may subsequently be used as the parameter of SET.O.REC (or SET.I.REC for an IO stream).

20) O.POS () / INTEGER32

This procedure returns a 32-bit index to the current character of the current output record. This may be used subsequently as a parameter to SET.O.POS.

21) O.SIZE () / INTEGER32

This procedure returns, as a 32-bit integer, the size in bytes of the current output stream.

22) O.DOC () / ADDR [LOGICAL8]

This procedure returns the name and type of the current output document. Additional information about the document type is returned in PW1, PWW1 and PWW2. PW1 is set to the type of document and determines the interpretation of the procedure result and PWW's, as follows:

	PW1	Procedure result	PWW1	PWW2
Current file	0	Nil	0	0
File	1	Pointer to filename	Directory name	0
Process	2	Nil	Process name	Machine name
Reply stream	3	Nil	REPn	0
Scratch document	5	Nil	0	0
Procedure	7	Nil	0	0
Module document	8	Pointer to module>document name.		

23) BREAK.OUTPUT (INTEGER)

This procedure terminates the current section of the specified output stream, and dispatches it to its destination. P1 specifies the output stream to be broken, where -1 indicates that the currently selected stream is to be broken. In the case of breaking output on streams other than message streams, further attempts to output to the stream will generate a trap (OUTPUT EXCEEDED). BREAK.OUTPUT has no effect on an undefined stream.

24) END.POS ()

This procedure defines the current record position in the current output stream to be its end. Its normal use is in truncating an IO stream.

25) OUT.HDR (ADDR [LOGICAL8])

This procedure replaces the header of the current output stream by the vector given by P1. The character string is *not* copied, so having called OUT.HDR subsequent, changes to the vector P1 will result in corresponding changes when the output is printed.

4.2 BASIC INPUT/OUTPUT OPERATIONS

The procedures described in this section provide the basic operations, in terms of which all other higher level input/output is implemented.

As explained in Chapter 3, both sequential and random operations are available. Sequential input/output always advances to the next position, while random operations are achieved by first moving the pointer to a specified position and then performing the corresponding sequential operations.

Three types of positioning operations are available, namely for positioning at the start of a specified record, for positioning within a record, and for positioning at the end of the last written record in an IO stream. These are as follows

1) SET.I.REC (INTEGER32)

This sets the starting position for a record within the current input stream. The parameter P1 is interpreted as a byte position, as returned by I.REC and O.REC. It is intended for higher level library organisations in implementing more complex file organisations.

2) SET.O.REC (INTEGER32)

This sets the starting position for a record within the current output stream. The parameter P1 is interpreted as a byte position, as returned by O.REC and I.REC. It is intended for higher level library organisations in implementing more complex file organisations.

3) WIND ()

This command should only be used when the current output stream is a IO stream. It positions the output pointer after the last record in the output so that further records may be written or the last record extended.

4) SET.I.POS (INTEGER32)

This sets the current position within the input record. The parameter P1 is interpreted as a byte position, as returned by I.POS and O.POS, with the start of the record denoted by a position of 0.

5) SET.O.POS (INTEGER32)

This sets the current position within the output record. The parameter P1 is interpreted as a byte position, as returned by O.POS and I.POS, with the start of the record denoted by a position 0.

4.2.1 Character Input/Output Procedures

The main operations for accessing a stream are IN.CH and OUT.CH. Additional procedures are NEXT.CH, which inspects the next character on the stream without actually advancing the pointer, and IN.BACKSPACE and OUT.BACKSPACE which provide limited facilities for backspacing in the input and output streams respectively. There are also procedures to enable larger quantities of data, such as complete lines, to be processed at a time.

1) IN.CH () / INTEGER

This procedure yields as its integer result the next character on the currently selected input stream, and the input pointer is advanced. An error is generated if the stream is undefined, or the last character of the stream (or of the current record) has already been read.

2) NEXT.CH () / INTEGER

This procedure has the same effect as IN.CH, except that the input pointer is not advanced. Thus many consecutive calls to NEXT.CH will yield the same result. The next call to IN.CH will also yield the same result.

3) IN.BACKSPACE (INTEGER)

This procedure backspaces the input pointer by an amount specified by P1. If P1 is negative, the pointer is moved to the start of the current line of text. This can still be used even after a line feed or formfeed has been read. For positive P1, the pointer is moved back P1 characters, or to the start of the current section, whichever is nearer.

4) IN.BIN (INTEGER) / LOGICAL32

This procedure reads the number of bytes specified by P1, yielding a right justified logical 32 result

5) SKIP.LINE ()

This procedure advances the current input stream to the start of the next line.

If the pointer for the stream is currently positioned at the end of a line – i.e. a linefeed or formfeed has been read, the input pointer will not be advanced.

6) IN.LINE (ADDR [LOGICAL8]) / INTEGER

This procedure copies a line of input from the current input stream into the buffer specified by P1, and returns as its result the number of characters in the line. The linefeed or formfeed which terminates the line is copied into the buffer, but is not included in the character count.

If the pointer for the stream is currently positioned in the middle of a line i.e. the linefeed or formfeed has not yet been read – the remaining characters (if any) in the line are copied.

7) IN.VEC (ADDR INTEGER)

This procedure copies characters from the current input stream into the vector whose address is given by P1. P2 specifies the number of characters to be read.

8) OUT.CH (INTEGER)

This procedure outputs its parameter as the next character of the currently selected output stream. In the case of a short message stream, if the current section buffer is full, BREAK.OUTPUT is called first.

9) OUT.BACKSPACE (INTEGER)

This procedure backspaces the output pointer by an amount specified by P1. If P1 is negative, the pointer is moved to the start of the current line. For positive P1 the pointer is moved back P1 characters, or to the start of the current section, whichever is nearer.

10) OUT.BIN (LOGICAL32, INTEGER)

This procedure outputs the number of bytes of P1 specified by P2

11) OUT.LINE (ADDR [LOGICAL8], INTEGER)

This procedure outputs a line, contained in the character string P1, to the currently selected output stream.

P2 specifies the carriage control to be used, and is to be interpreted as follows:

```

0      -- overprint previous line (i.e.  output carriage
      return before printing).
1      -- print on next line (i.e.  output one linefeed
      before printing).
2-63   -- leave specified number (1 -62) of blank
      lines before printing (i.e.  output linefeeds
      first).
64     -- print at start of next page (i.e output one
      formfeed first).
65     -- output one linefeed after printing.
66-127 -- output 2-63 linefeeds after printing.
128    -- output formfeed after printing.
```

12) OUT.VEC (ADDR, INTEGER)

This procedure copies characters to the current output stream from the vector whose address is given by P1. P2 specifies the number of characters to be output.

4.2.2 Record Input/Output Procedures**1) IN.REC () / INTEGER32**

This procedure selects the next record in the currently selected input stream, returning as a 32-bit result the size of the record, or -1 if there are no more records. Subsequent sequential character or binary operations will operate within this record.

2) OUT.REC ()

This procedure completes the current record on the currently selected stream, and starts a new record.

4.3 OTHER (CHARACTER STREAM) INPUT/OUTPUT PROCEDURES

For the majority of user's, the input/output interface is defined by the programming languages they use, and is implemented in terms of the basic procedures described in Section 4.2. However, there are some further procedures available in the library for doing common character input/output operations (such as reading and printing decimal integers). These are used in the implementation of the basic system, and in effect form the input/output interface of the system programming languages, but they may also be used by programs written in other languages.

4.3.1 Character Stream Input Procedures**1) IN.I () / ADDR**

This procedure reads a (possibly signed) decimal integer from the selected input stream, returning the value as its result. A trap is forced if the next nonblank character is not a decimal digit, + or -.

2) IN.OCT () / INTEGER

This procedure reads an octal integer from the currently selected input stream, returning the value as its result. A trap is forced if the next nonblank character is not an octal digit.

3) IN.HEX () / LOGICAL64

This procedure reads a hexadecimal number from the currently selected input stream, returning a LOGICAL64 value as its result. A trap is forced if the next nonblank character is not a hexadecimal digit.

4) IN.REAL () / REAL64

This procedure reads a number and yields a real value. The number read consists of an optionally signed integer or a real, followed by an optional exponent. The exponent consists of a 'E', 'e' or '@' followed by an optionally signed integer.

5) IN.C.LIT () / LOGICAL64

This procedure reads a string of characters enclosed in double quote symbols from the currently selected input stream, returning the string as a packed, right-justified LOGICAL64 value. If too many characters are given, the required number are taken from the end of the string. Non-printing characters can be represented by using their ISO character codes written as two hexadecimal digits and enclosed between exclamation marks, e.g. "ABC". "A VERY LONG STRING", "!OC!PAGE3". A trap is forced if the next character is not ".

6) IN.NAME () / LOGICAL64

This procedure is similar to IN.C.LIT except that the string should not be enclosed in quotes, preceding blank lines and spaces are ignored, and the string terminates on reading either a space or a newline. (This procedure is commonly used in the system for reading filenames, etc.)

7) IN.C.STR (ADDR [LOGICAL8]) / INTEGER

This procedure reads a character string enclosed in double quotes from the currently selected input stream, in the same format as for IN.C.LIT

above. The string is placed in the byte vector described by P1, and its size (number of characters) is returned as an integer result. If the size of the byte vector is too small to accommodate all the characters given, any remaining characters are ignored up to the end of the string.

8) **IN.STR (ADDR [LOGICAL8]) / INTEGER**

This procedure is similar to **IN.C.STR** except that the string should not be in quotes, preceding blank lines and spaces are ignored, and the string terminates on reading either a space or a newline. (This procedure is used by the job control interpreter to read string parameters.) Notice that spaces within the string must be represented by their hexadecimal character codes. e.g.

A!20!STRING!20!WITH!20!SPACES.

4.3.2 Character Stream Output Procedures

1) **SPACES (INTEGER)**

This procedure outputs the number of spaces specified by P1 to the currently selected output stream.

2) **NEW.LINES (INTEGER)**

This procedure outputs the number of newlines specified by P1 to the currently selected output stream. If P1 is zero and the previous character on this stream is newline, nothing is output; otherwise a single newline is output.

For short message streams (e.g. online output), **BREAK.OUTPUT** is called to dispatch the current section on each call to **NEW.LINES**.

3) **CAPTION (ADDR [LOGICAL8])**

This copies the string of characters in the byte vector P1 to the currently selected output stream.

4) **OUT.I (INTEGER32, INTEGER)**

This procedure prints the integer P1 as a decimal integer on the currently

selected output stream. P2 specifies the required field width. If $P2 = 0$, the number is printed left-justified. Otherwise, it is printed (if possible) in a field width of P2 characters, preceded by a space for positive numbers, a minus sign for negative numbers (P2+1 characters in total). If the number requires more than P2 digits, it will overflow the specified field width.

5) OUT.HEX (LOGICAL32, INTEGER)

This procedure prints the parameter P1 (which is a LOGICAL32) as a P2-digit hexadecimal number on the currently selected output stream. P2 should be in the range 1 to 8. If P2 is zero, only significant digits are printed.

6) OUT.REAL (REAL64, INTEGER, INTEGER)

This procedure prints the real number in P1. P2 specifies the total number of characters to print while P3 specifies the format of the number.

$P3 \neq 0$ specifies the number of decimal places to print. The format of the output consists of a ‘-’ (if the number is negative), an integer part, a ‘.’ and finally the fractional part.

$P3 = 0$ If P2 is zero a default width of 10 is used. The format of the output consists of a ‘-’ (if the number is negative), ‘0.’ followed by the . fractional part and the exponent in the form $E \pm \text{integer}$. At least eight characters will always be printed, even if a smaller field width is specified.

7) OUT.TIME ()

This procedure prints the time, in the format HH:MM:SS (hours, minutes, seconds) on the currently selected output stream

8) OUT.DATE ()

This procedure prints the date, in the format DAY-MONTH-YEAR (e.g. 01-APR-84) on the currently selected output stream.

9) OUT.TD (LOGICAL64, INTEGER)

This procedure prints the time or date, extracted from its parameter P1.

This parameter is assumed to be a 32 bit integer, as returned by the system procedure `TIME.AND.DATE` (see Chapter 8). If P2 is zero, the time is printed in the same format as for `OUT.TIME`. If P2 is non zero, the date is printed in the same format as for `OUT.DATE`.

10) `OUT.LINE.NO (INTEGER32)`

This procedure prints the packed page/line number specified by P1 on the current output stream. The page and line numbers are printed in decimal, separated by '.' with a total field width of 10 characters. The most significant 16 bits of P1 should give the page number and the rest of P1 gives the line number.

11) `OUT.NAME (LOGICAL64)`

This procedure outputs its parameter as eight characters, left justified, with spaces to the right replacing any leading null characters.

12) `OUT.FN (ADDR [LOGICAL8])`

This procedure is used by compilers etc., to print out 'name time date' at the head of compilations, file listings, etc. P1 is the name to be printed. If it is zero, the name `<CFILE>` will be printed.

13) `OUT.MACHINE ()`

This procedure prints identification information about the and operating system on the current output stream. The identification is in the form

Machine name (Machine number / V operating system version)

14) `ECHO.LINE ()`

This causes the current line of input to be printed on the current output stream if the output stream is offline. Otherwise, it has no effect.

15) `PROMPT (ADDR [LOGICAL8]) / ADDR [LOGICAL8]`

This procedure resets the system prompt message, so that the specified string (P1) is used the next time the system prompts an online user for input. The prompt string will be reset by the job control interpreter next time it

regains control. If the parameter is zero, prompting is suppressed completely. The procedure returns a pointer to the previous prompt message.

The prompt message is *not* copied, so subsequent changes to the prompt string P1 will result in corresponding changes to the output when the prompt message is printed.

16) PROMPT.CH (INTEGER)

To allow the interception and recognition of prompt messages, a prompt warning character is output immediately preceding the prompt message. This warning character is set using the PROMPT.CH command. If the prompt warning character P1 is zero, the warning character is omitted from prompt messages.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 5

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

5 FILES AND EDITING

5.1 GENERAL DESCRIPTION OF THE FILE SYSTEM

The file system provides users with the means of retaining information inside the system, and of accessing and altering this information from within jobs. Often, though not always, the files are created by programs or system commands, using the input/output facilities described in Chapters 3 and 4.

Each user of the system has a file directory, which lists the files owned by that user. Also it is possible to create a hierarchy of directories, which have a tree structure with the users original directory as the root. As described in Section 2.4.1 the name of a directory or file is specified to the system as a pathname, and the terms *filename*, *local filename*, *directory name* and *local directory name* were defined. A single file may contain up to 256K bytes of information (text files may be larger), but individual users are restricted both in the number of files they may own and in the total amount of space their files may occupy. If a user exceeds either of these limits, further attempts to access the files will be prevented until the usage is brought back within the allocated limits either by deleting files or by increasing the user's allocation.

A user does not normally need to be aware of the exact location of his files, as this is managed automatically by the system. There is one aspect of this, however, which may concern the user. The file system can extend across up to three levels of storage, and files are automatically transferred by the system between the three levels as necessary. Files which are currently in use or have recently been accessed reside in the local filestore. Files which have not been used for some time may be automatically offloaded to a large capacity file buffer, from which they may be further removed to archive storage on removable storage media (e.g. magnetic tapes). The user need not be aware of this movement, since files will be retrieved automatically into the local filestore when they are accessed, but retrieving a file from the archives may involve an appreciable delay. Also, commands are provided for the user explicitly to request file offloading and onloading, for security reasons. This mechanism is only provided on machines with a suitable hardware configuration.

5.2 SHARING FILES

As has been mentioned in 2.4.1, it is possible for users to reference the files of other users. This depends upon two mechanisms. First, the users are organised in a hierarchy which starts from the user SYSTEM. Any user may access the (files of a subuser without the need for special permission. The second mechanism allows more selective sharing of files across the hierarchy of users. Any user may permit others selectively to share his files by calling on the PERMIT command, giving the access permission to be granted to each user.

5.2.1 Network File Systems (not applicable to stand alone systems)

Each machine in a MUSS network will have its own independent file system. However, the file commands are such that users are able to open directories and make use of files in other machines. This involves passing messages between the different file systems. To specify the necessary information about the required directory in another machine, a command NAME.DIR is provided. This assigns a pseudo username for the required directory, which may subsequently be used as the username parameter of file commands to access the remote files.

It is essential that users have been granted the necessary permission to access the files in the remote machines.

5.3 FILE SYSTEM COMMANDS

In the following file system commands, whenever the user whose file store is to be accessed is *not* specified, the local file/directory name will be assumed to be relative to the currently selected directory of the user.

1) OPEN.DIR (ADDR [LOGICAL8])

This command allows a user to select a subdirectory to be used instead of his main directory, as the starting point of paths in subsequent *local filenames* and *local directory names*. The effect is the same as if the selected directory was the root directory, except for path names which begin with ‘/’ which always take the user’s main directory as their root. P1 is the *local directory name* of the required directory. If P1 is the null string the main directory of

the user is restored as the root.

2) `OPEN.FILE (ADDR [LOGICAL8], LOGICAL64, INTEGER, LOGICAL)`

This command opens a file with the *local filename* P1 into the virtual store of the current process P2 is the username of the file's owner with a default of zero implying the current user P3 and P4 specify a segment and access permission for the file. If the segment number is negative, a free one will be allocated by the system. The access permission is formed from the six segment access bits (see Chapter 13) combined with the next most significant bit, which if set indicates that exclusive access is required. If the Task copy bit (%20) is set the file is copied into a scratch segment, which then has no further association with the file. Any access permission is allowed for files in the current directory, but for files borrowed via the PERMIT facility, the calling process will be faulted if the specified access exceeds that stated in the PERMIT file. PW1 returns the number of the segment holding the file, and PW2 and PW3 the size and status of the segment respectively.

3) `FILE (ADDR [LOGICAL8], LOGICAL64, INTEGER)`

This command preserves a segment (P3) as a file with *local filename* P1. P2 is the name of the user who is to own the file, with a zero default implying the current user. The segment remains in the process' virtual store after a file operation. A segment which was created by an `OPEN.FILE` command cannot be used in a `FILE` command unless exclusive use was requested by the `OPEN.FILE` command.

4) `DELETE.FILE (ADDR [LOGICAL8], LOGICAL64)`

This command deletes the file with *local filename* P1 belonging to user P2. P2 may be zero implying current user.

5) `RENAME.FILE (ADDR [LOGICAL8], LOGICAL64, ADDR [LOGICAL8])`

This command is used to rename the file with *local filename* P1 belonging to user P2. P2 may be zero implying current user. P3 gives the new file name which must be a simple name not including ':' or '/'.

6) `CATALOGUE.FILES (ADDR [LOGICAL8], LOGICAL64)`

This command creates a copy of the directory, specified by the local directory name P1, in a newly created segment. PW1 returns the segment number. P2 is the username of the directory owner, with a zero default implying the current user. The segment consists of an area containing data referring to the whole directory, followed by directory entries describing individual sub-directories or files (sub-directories first then files).

If P2 is zero or specifies the current user a list of *all sub-directories and files*, held in the directory P1, is returned, otherwise *only files* permitted to the current user (via the permit command) are returned. Each directory entry in the segment is 40 bytes long and takes the following form

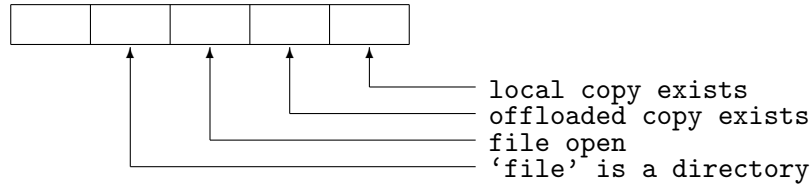
File or Directory Name (16 bytes)
 Status (32 bits) (not used if sub-directory)
 Size in bytes (32 bits)
 Update time in seconds (32 bits)
 Create time in seconds (32 bits) (not used in sub-directory)
 Disc address (32 bits)

The initial area of the segment before the directory entries is 56 bytes long and contains a 16 byte directory name, a 64 bit username and eight 32 bit integers. These contain the following information.

- i) The number of subdirectory entries in the segment
- ii) The number of file entries in the segment
- iii) The amount of filestore (in bytes) used in the directory
- iv) The number of free entries for use in the directory
- v) The total number of files owned by the user
- vi) The total amount of filespace (in bytes) owned by the user
- vii) The limit on number of files for the user
- viii) The limit on filespace (in Kbytes) for the user.

7) READ.FILE.STATUS (ADDR [LOGICAL8], LOGICAL64)

This command reads the directory information associated with the *local filename* P1 in the directory of user P2. PW1 returns the status information in the form:



PWW1 holds the size of the file in bytes, and PWW2 and PWW3 return the update time and the create time (in seconds) respectively.

8) PERMIT (ADDR [LOGICAL8], LOGICAL64, LOGICAL64, LOGICAL64)

This command may be called by a user to assign a set of access rights to one of his files for another user. P1 specifies the local filename and P2 the name of the user to be granted permission. The option 'ALL' is available for both parameters, to allow a user access to all the files in the current directory, or allow all users access to a file.

The third parameter gives the permissions assigned, and is specified as a combination of the letters:

X	meaning permission to open with execute access
W	meaning permission to open with write access
R	meaning permission to open with read access
E	meaning permission to open with exclusive access
C	meaning permission to change access when open
U	meaning permission to update the file
D	meaning permission to delete the file
N	meaning permission to rename the file

The letters may appear in any order.

To remove access permission rights, the PERMIT command should be called with P3 zero.

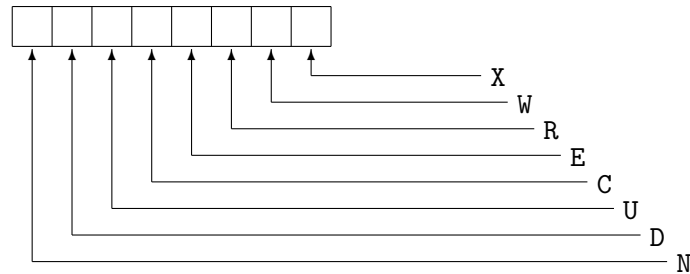
If P4 is non-zero it should be a name previously defined by a NAME.DIR command. In that case the PERMIT command will apply to the files of the username given in the NAME.DIR command instead of the current user.

9) CATALOGUE.PERMIT (ADDR [LOGICAL8], LOGICAL64)

This command creates a copy of the permissions associated with the directory specified by the *local directory name* P1. P2 is the username of the directory owner, with a zero default implying the current user. If P2 is zero or specifies the current user a copy of *all* permissions associated with the directory P1 is returned, otherwise only those permissions granted to the current user are returned. A segment is created for this purpose, and the segment number is returned in PW2. The permission entries are of the form

FILENAME (16 bytes)
 USERNAME (8 bytes)
 ACCESS (1 byte)

The access is bit significant, and has the format



The number of entries in the segment is returned in PW1.

10) NAME.DIR (LOGICAL64, LOGICAL64, LOGICAL64, ADDR [LOGICAL8])

This command allows pseudo usernames to be used in place of usernames including those in other file systems. Thus it can be used to cause the normal file commands to apply to other file systems controlled by file manager processes. It introduces the name P1 which may subsequently be used instead of a username in other file commands, and in *filenames* and *directory names*. P2 names the file manager process required, and P3 names the machine on which it runs. If P2 is zero the default name "FILMAN" will be used. P4 is a string containing additional information required by the particular file manager. Examples of the use of this command can be found

in Section 5.3.2.

11) RELEASE.DIR.NAME (LOGICAL64)

This command cancels the effects of a previous NAME.DIR command. P1 is a username introduced by a NAME.DIR command. After this procedure is called this username will no longer be recognised as a reference to a file manager process. Also a message will be sent to the file manager process informing it that any resources reserved by the NAME.DIR command can now be released.

12) CREATE.SUBDIR (ADDR [LOGICAL8])

This command creates a new subdirectory, with local directory name P1, within the current directory it becomes part of the current tree of directories. P1 must be a simple name not including ':' and '/'.

13) DELETE.SUBDIR (ADDR [LOGICAL8])

This command deletes a subdirectory of the current directory. P1 is the local directory name of the subdirectory. P1 must be a simple name not including ':' and '/'. There must be no files or further subdirectories within the specified subdirectory.

14) CREATE.FILE.SEGMENT (INTEGER, ADDR, ADDR [LOGICAL8], LOGICAL64)

This command is equivalent in effect to the CREATE.SEGMENT command described in Chapter 13. It is used when a segment is created with the intention of later filing it away as a particular named file. By giving the filename and username information at the time of creation the system can optimise the allocation of resources to the segment. The parameters P3 and P4 give the local filename and username of the eventual file. The parameters P1 and P2 and the result returned in the PW variables are exactly as described for CREATE.SEGMENT.

15) CREATE.FILE.X.SEGMENT (INTEGER, ADDR [LOGICAL8], LOGICAL64)

This is identical to CREATE.FILE.SEGMENT except that the command is equivalent to CREATE.X.SEGMENT not CREATE.SEGMENT.

16) CHANGE.ROOT.DIR (LOGICAL64, LOGICAL64)

This procedure changes the main root directory to that of another user. P1 and P2 are the username and password of the new user. After using this procedure all file operations will work as if the current process was that of the new user. If P1 and P2 are given as zero the root directory of the user name used to login will be selected.

17) CHANGE.FILE.SIZE (ADDR [LOGICAL8], LOGICAL64, INTEGER)

This procedure changes the size of a file. P1 is the local filename and P2 the user. P3 is the required file size in pages. (The size of a page depends on the specific file hardware and file manager.) The file must not be open when this command is used. In changing the size the procedure may convert the file into an X-segment.

5.3.1 Other File and Current File Commands

These commands allow the user to manipulate files and set up the current file. To access another user's files, the form "username:filename" should be used.

- 1) OLD (ADDR [LOGICAL8]) This command designates a file, specified by the *filename* P1 as the current file.

- 2) SAVE (ADDR [LOGICAL8])

This command preserves the current file as a permanent file. P1 gives the *filename*. If a file of this name exists already, it will be replaced. The file continues to be the current file.

- 3) DELETE (ADDR [LOGICAL8])

This command is used to erase a file whose *filename* is given by P1. If P1 is zero or omitted, the current file is deleted.

- 4) RENAME (ADDR [LOGICAL8], ADDR [LOGICAL8])

This command is used to change the name of an existing file. P1 specifies the old filename. P2 specifies the new name, which may not involve ':' or '/' and will be assumed to be in the same directory as

the old *filename*. If the new file name already exists in the directory, an error is signalled.

- 5) `LIST.DIR (ADDR [LOGICAL8], ADDR [LOGICAL8], ADDR [LOGICAL8])`

This command lists, on the currently selected output stream, the files in a specified directory

P1 is a character used to indicate the level of detail required in the directory listing, as follows:

P1 = 0 File limits and list of file names.
P1 = 'N' File limits and list of file names.
P1 = 'L' File limits only
P1 = 'A' All information.

P2 may be used to select a subset of the files in the directory. If P2 is non-zero, only files whose names contain the string P2 will be listed.

P3 is a *directory name* which may be omitted if the current directory is required.

- 6) `FIND.FILES (ADDR [LOGICAL8], ADDR [LOGICAL8])`

This command creates a list on the current file of the filenames in the specified directory. P1 is a directory name which may be omitted if the current directory is required. P2 may be used to select a subset of the files in the directory. If P2 is non-zero, only files whose names contain the string P2 will be listed.

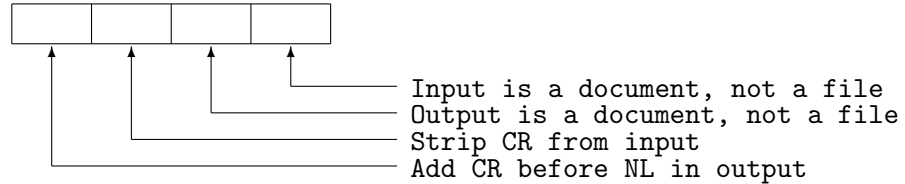
- 7) `LIST.FILE (ADDR [LOGICAL8], ADDR [LOGICAL8])`

This command is used to list a text file. P1 specifies the document to be listed, and P2 the destination document. If P2 is zero or omitted, the listing is on the currently selected output stream.

- 8) `COPY.FILE (ADDR [LOGICAL8], ADDR [LOGICAL8], LOGICAL);`

This command copies the data in the file or text document P1 into the file or document P2. P3 is a mode parameter which determines

whether each of the first two parameters is treated as a file or is a document, i.e., whether the data is copied to or from a raw binary image or a MUSS I/O document. The format of P3 is as follows:—



Note that the CR modification bits have no effect when both input and output are files.

For example,

- 1) to read a file (filename) from a 40 track PC disc into a MUSS document (DOC1) using a 80 track double density drive

```
NAMEDIR X IBM 0 (PC-R/)
CF X:filename DOC1 6
```

note: the volume name checking is optional as on the PC.

- 2) to transfer a MUSS document (DOC2) to a file (filename) on a standard 40 track PC disc using a 40 track PC compatible drive

```
NAMEDIR X IBM 0 (PC/)
CF DOC2 X:filename 9
```

note if 80 track drives are used in 40 track mode the disc may not be readable on 40 track drives.

- 9) EDIT (ADDR [LOGICAL8], ADDR [LOGICAL8])

This command invokes a simple line editor, with limited screen editing capability, to modify a text file. A complete specification of this and other editors appears in Section 5.4.

- 10) LIST.PERMIT (ADDR [LOGICAL8], ADDR [LOGICAL8], ADDR [LOGICAL8])

This command lists, on the currently selected output stream, the file permissions associated with a file directory.

P1 may be used to select a subset of the permissions. If P1 is non-zero, only files whose names include the string P1 will be listed. Similarly, P2 may be used to select a subset of the usernames for whom permissions have been granted. P3 specifies a directory *name*. If P3 is not given the current directory will be assumed.

- 11) COMPARE (ADDR [LOGICAL8], ADDR [LOGICAL8], INTEGER)

This procedure compares the documents given by P1 and P2, monitoring the differences on the current output stream. The format of the output is controlled by P3. If P3 is zero, both versions of the lines which are different are printed. If P3 is non-zero, the file P1 is listed with the relevant lines marked with a ! at the start of each line that is different.

5.3.2 File Manager Processes

As mentioned in the description of NAME.DIR above, the services provided by file manager processes can be utilised through normal file commands. Certain such processes are commonly present in MUSS systems, and these are described below. The descriptions give the process name (P2 of NAME.DIR) and then explain the use of the initial information (P4 of NAME.DIR). A value of 0 for P2 implies the FILMAN process of the machines specified by P3.

- 1) FILMAN

This process provides access to the standard file store for other machines in the network. It also allows additional flexibility in accessing files in the same machine. The information needed by FILMAN to access a file is a string of form: –

(username/password)file name.

This string can be arbitrarily split between the initial information

passed as P4 of NAMEDIR and the file name parameter of subsequent file commands.

The username and password should be valid ones for the machine on which the FILMAN process is running.

The "file name" is a full file name optionally containing another username. If this second username is present access to files will be determined by the permissions granted to the user identified by the "(username/password)".

2) FLOPPY

This process is used to handle files on low capacity, easily exchangeable media such as floppy discs. The file store structure is kept simple, with no division into users or directories. Apart from this restriction all the normal file commands are usable. The initial information passed as P4 of NAMEDIR identifies the disc and ensures that the user has access to it. It is in the form:—

(discname/password)

The discname and password are the volume name and label, as described in Chapter 17.

If the discname IS prefixed with "CLEAR:" it will have its file store data structure initialised.

The filenames used in the file operations with this file manager should be a single name not a path name.

3) IBM

This process allows the files and directories on IBM or IBM PC/AT compatible floppies to be manipulated by the normal file commands.

The floppy drive may be one of the following three types

- 1) A 40 track double density drive (IBM PC compatible)

- 2) A 80 track AT high density drive
- 3) A 80 track double density drive.

The string passed in P4 of NAMEDIR should be of the form

(floppytype/volumename)

and it is mainly used to indicate the drive type

- 1) PC (Read/write 40 track IBM PC floppy)
- 2) AT (Read/write 80 track IBM PC/AT floppy)
- 3) PC-R (Read an IBM PC floppy)

In cases 1) and 2) there are no restrictions on file operations. Case 3) allows 40 track floppies to be read on an 80 track double density or IBM PC/AT compatible drive.

If a volume name part is given it is compared with the PC-DOS disc volume name and an error is returned if they are different.

If the floppy type is prefixed with 'CLEAR:' the PC-DOS data structure on the disc will be initialised.

The filenames used in the file operations with this file manager should be PC-DOS path names with '/' used as the directory separator

5.4 FILE EDITING

Several editors are normally available in MUSS as private libraries. The built-in one described here is a line editor with screen editing facilities. It enables users to make alterations to files of text, and it is invoked by the command

EDIT (ADDR [LOGICAL8], ADDR [LOGICAL8])

Both parameters of this command are filenames (or more accurately, document names as described in Chapter 3). The first is the input file, and the second the output file, and the action or the editor is to make an altered

copy of the input file on the output file. The two filenames may be the same, in which case the input file is overwritten by the output when the edit is ended (or the T command is used). In general, the two names may take any of the forms described for input and output documents in Section 2.4. except that unbuffered (online) documents may not be used.

When the editor is called from an online terminal it will automatically enter screen editing mode. If it is called from a command stream it will enter line editing mode. These two modes are described separately in the following Sections. However, facilities exist for the user to change from one mode to the other.

5.4.1 Screen Editing Mode

The behaviour of the editor in this mode is very similar to the “structured document” editor in Section 1.6.3. In summary the cursor movements are controlled by the move left ($\leftarrow$), right ($\rightarrow$), up ($\uparrow$) and down ($\downarrow$) keys. Most other actions are determined by control characters as follows

$\uparrow$ A	Move back to the start of the file
$\uparrow$ B	Search backward for the string which follows
$\uparrow$ B	Repeated will scroll half a page back
$\uparrow$ D	Delete the next character
DEL	Delete previous character
$\uparrow$ E	End the edit
$\uparrow$ F	Search forward for the string which follows
$\uparrow$ F	Repeated will scroll half a page forward
$\uparrow$ I	Insert spaces up to next tab position
$\uparrow$ R	Recover deletion
$\uparrow$ T	Duplicate line
$\uparrow$ U	Move forward to the next word
$\uparrow$ W	Delete up to next space (1 word)
$\uparrow$ X	Delete rest of line
$\uparrow$ Z	Move forward to next page.

The differences between this and the document editor are mainly concerned with $\uparrow$ A, $\uparrow$ Z, $\uparrow$ L, $\uparrow$ O and $\uparrow$ V. The effect of $\uparrow$ A is to save the changes already completed in the destination file, and restart the edit again from the beginning. In the structured document editor the use of $\uparrow$ Z relates to

the fact that structured documents are formally divided into a sequence of records called pages. Operation of $\uparrow Z$ moves to the next 'page' and the display changes accordingly. In the case of this editor the files are not formally subdivided, therefore a 'page' is taken to be an actual page, i.e. a sequence of text delimited by formfeed characters. If there is a large amount of unpaginated text the editor will make its own arbitrary subdivisions. In either case the whole of a large file can only be accessed by using $\uparrow Z$ to move forwards a page at a time and $\uparrow A$ to return to the beginning of the file. Remember that $\uparrow A$ secures the changes done up to that point. All of the remainder ($\uparrow L$, $\uparrow O$ and $\uparrow V$) have the same effect. They cause the editor to change from screen editing mode to line editing mode, with the next line being the beginning of the next page.

A special mention needs to be made of long lines, i.e. lines too big to fit on one line of a VDU terminal. When such lines exist in the page which is currently displayed, they are broken into several lines. This is done after 77 characters, by the inclusion of a '|' character followed by a newline character. Regardless of line length, if any line on a page ends with the '|' character, then the '|' and the following newline will be removed. This takes place on moving to another page or on completion of the edit.

5.4.2 Line Editing Mode

The operation of the editor in line mode is controlled by means of a sequence of commands on the command input stream. Each command is identified by a single letter which may be followed by a parameter. Any number of commands may be placed on a line; blank lines and spaces occurring between commands are ignored. Edit commands are accepted in upper or lower case. A variety of commands for copying, skipping and inserting are provided, allowing different ways of specifying exactly what is to be copied, skipped or inserted. If the editor is in online mode and a fault occurs it will print a fault message and then ignores any further commands on the same line. If the command in which the fault occurred was an editing command (S, C, A, B, D) then the input and output pointers are restored to their values at the start of the command.

The use of the editor can be defined in terms of three operations, namely

1. *Copying* information unchanged from the input file

- to the output file.
- 2. *Skipping* over information in the input file, without copying to the output file.
- 3. *Inserting* new information into the output file.

The editor requires *lines* to be terminated by the linefeed character, and *pages* to be terminated by the formfeed character (Note that this use of the term page has nothing to do with the page units of storage allocation).

Most parameters are straightforward (e.g.. an integer) and they require no special explanation. However, a number of commands have as their parameter a string. A string is simply a sequence of characters between a pair of string delimiters. Any single non-alphanumeric character which does not appear in the sequence of characters may be used as a string delimiter. Thus, any of the following are legal strings:—

```
'ABC DEF'
/'BEGIN'/
$'BEGIN'
'INTEGER' I, J, K;
I := 1/2;
$
```

Sometimes it may be necessary to include as part of a string a character which cannot be typed on the input device, (for example a formfeed character). This may be achieved by typing the code for the required character as two hexadecimal digits enclosed between two string delimiters. Thus, for example using the ISO character codes:—

- 1. `!'BEGIN'!09!'INTEGER' I,J,;!` means 'BEGIN' followed by a tab character followed by 'INTEGER' I,J,;
- 2. `!!OC!TITLE!` represents a formfeed character followed by TITLE
- 3. `//OA//OA//` represents two consecutive newline characters.

Note that this mechanism is restricted to characters with codes in the range 00 – 9F. Certain characters are required so frequently that a special

representation exists within strings for them. This consists of a single letter enclosed between a pair of string delimiters. The characters for which a special representation exists are:–

Newpage (formfeed)(P) e.g., //P /TITLES/

Newline (L) e.g., //L//L//

Note: Users should be very careful in using the hexadecimal representations, particularly in insertions, as it is possible to introduce non-standard non-printing characters into the output file. These may then be hard to identify in subsequent editing. Also there may be unexpected effects when using line-editing commands (see later).

This same mechanism is used to introduce a special character in search strings. The effect of this character (?) is to cause a match on any character in the text string. Thus,

C/AB/?/DE/

will match any string that starts with AB and has DE in the fourth and fifth character positions, i.e., ABXDE. See Section 5.4.2.2 for the description of the commands, such as C, which use this string form.

5.4.2.1 Brief Summary of Available Commands

A brief description of the available commands is given below. They will then be described individually in greater detail. There are 9 commands altogether, and they are grouped for ease of description into 3 categories. Commands from all 3 categories may, however, be freely interspersed.

A. Line-editing Commands

1. S (SKIP) Skips to the start of a specified line
2. C (COPY) Copies to the start of a specified line
3. I (INSERT) Inserts a string into the output stream.

B. Context editing Commands

1. B (BEFORE) Copies up to the start of the next occurrence of a specified string.

2. A (AFTER) Copies up to the end of the next occurrence of a specified string.
3. D (DELETE) Copies up to the start of the next occurrence of a specified string, then skips to the end of the string (thus deleting it).

C. Other Commands

1. W (WINDOW) Causes a section of the file being edited to be listed on the monitor stream.
2. R (RESTORE) Causes the input and output pointers to be restored to the values they had before the previous line or context editing command.
3. T (TOP) Copies all remaining characters from the input stream to the output stream, files the output. and reopens the output as input.
4. E (EXIT) Copies all remaining characters from the input stream to the output stream, and returns control from the editor
5. M (MERGE) Alters the editor input file
6. Q (QUIT) Abandons the edit and returns from the editor without updating the output file.
7. V (VIEW) This re-enters screen editing mode.

5.4.2.2 Line Editing Commands (S, C, I)

These commands are used when whole lines are to be deleted, inserted or replaced. The two positioning commands (SKIP, COPY) always position the input pointer at the start of some specified line. As explained in next Section, SKIP merely advances the input pointer without copying any information to the output stream. COPY advances the input pointer, copying all characters it passes into the output stream. The INSERT command is not specifically a line-editing command, as the same command is used in both line and context editing. It is described separately in this section as the method of use for line editing is slightly different.

The SKIP and COPY commands enable the line required to be specified in a number of ways, as described below, (NOTE all positions are positions in the input stream, the output stream position being specified implicitly).

1. As a relative page number, e.g. `S + 5.`, meaning skip to the top (i.e., line 1) of page `i + 5` where `i` is the current page. Thus `S + 1.`, means 'skip to top of next page'
2. As a relative line number, e.g., `S + 20` meaning skip to the start of line `j + 20` where `j` is the current line. (Effectively, skip over 20 lines).
3. As an absolute page.line position e.g. `C5.4` means copy to the start of line 4 on page 5.
4. As a string, e.g., `S'BEGIN'` meaning skip to the start of the next line which contains the string `BEGIN`.
5. The letter 'F'. (e.g. `CF`) meaning copy (or skip) to the end of the file. Because skipping in this case is a dangerous command, the editor will always request verification before proceeding by printing 'REALLY?' If the response to this is 'Y', the editor proceeds; otherwise the command is abandoned as faulty. Thus in offline mode, to skip to the end of the input, the sequence 'SFY' is necessary.

Note that use of control characters such as newline in strings in the `SKIP` and `COPY` commands can have rather unexpected results. for example, consider.−

```
C//L/BEGIN/
```

or for that matter

```
C/  
BEGIN/
```

means copy to the start of the line containing the string 'linefeed' `BEGIN`. Thus if the file contained:−

```
END OF PROCEDURE;  
BEGIN
```

the pointer would finish at the start of the `END` line. If you don't understand this, never use control characters in `S` or `C` commands!!

Insertion

The Insert command has a string as its parameter, and inserts all characters between the string delimiters into the output stream. Since the SKIP and COPY commands always stop at the start of a line, this means that to insert a single line into the output the insert command should normally be used as follows:-

```
I/NEW LINE TO BE INSERTED
/
```

The following alternative:

```
I/
NEW LINE TO BE INSERTED
/
```

would insert the new line with an extra blank line before it and

```
I/
WRONG WAY OF DOING IT/
```

would insert an extra blank line and then simply insert the new string at the start of the existing line. This may of course be useful in some cases, e.g.,

```
S/INTEGER /I/
BEGIN/
```

will result in:

```
BEGIN INTEGER
```

NOTE: When inserting new pages, the formfeed character should always be placed on a new line Thus:

- (a) To split an existing page into 2 pages such that the second starts 50 lines from the current position

```
C+50I//P//
```

has the desired effect

- (b) To insert a completely new page after the current page

```
C+1.I
/FIRST LINE OF NEW PAGE
/LAST LINE OF NEW PAGE
/P//
```

Summary of line-editing Syntax (by examples)

1. S3.1 skip to top of page 3.
2. s + 10 skip over next 10 lines of input.
3. C'XYZ' copy to start of next line containing XYZ.
4. CF copy to end of input file.
5. SFY skip to end of input file
6. I normal single line insertion.
/LINE TO INSERT
/
7. I multiple line insertion.
/LINE 1
LINE 2
ETC
/
8. I page insertion (if initially positioned at
/LINE 1 start of page).
LINE 2
ETC
/P//
9. I//P/LINE1 page insertion (if initially positioned at
LINE 2 end of page).
ETC
/

5.4.2.3 Context-Editing Command (B, D, A, I)

These commands may be used to edit individual character:s or strings of characters within a line. All four commands have a single string parameter. The Insert command is actually the same command as the one used for line editing.

The ‘BEFORE’ command (B)

The ‘BEFORE’ command is basically a copy operation, copying characters from input to output until the input pointer arrives at the start of the next occurrence of a specified string. Unlike the line-editing copy operation (C), however, the input pointer is left immediately before the start of the string, not at the start of the line. For example, to correct:–

```
BEGIN INTER I,J,K;
```

the commands

```
B/ER/I/EG/
```

would be used to copy to immediately before the string ER and then insert EG. In making corrections of this type, of course, it is important to ensure that the string being searched for is unique. For example, B/E/I/EG/ would have resulted in:

```
BEGEGIN INTER I,J,K;
```

The ‘AFTER’ command (A)

This is also a copying operation, leaving the input pointer immediately after the next occurrence of the string. Thus for example, to correct:–

```
BEGIN INTE I,J,K;
```

the commands

```
A/INTE/I/GER/
```

could be used.

The ‘DELETE’ command (D)

The delete command is a combined copy and skip operation. Its action is to copy to the start of the specified string, and then skip to the end of the string, thus effectively deleting it from the output. This operation is especially useful for correcting spelling or typing errors, for example:–

BEGONE INTEGER I,J,K;

can be corrected by

D/BEGONE/I/BEGIN/

or, provided that the string ONE does not occur before the spelling mistake, by

D/ONE/I/IN/

The ‘INSERT’ command (I)

Examples of the use of the insert command have already been given in the preceding section. Its action, as in the case of line editing, is simply to insert all the characters in the string into the output stream.

Strings used in context-editing commands may, of course, contain newline and newpage symbols as in the line editing commands. The most common use of the context editing commands, however, is to correct small errors within a line. Care should be taken in context editing to avoid deleting the final newline character of a page or file.

Summary of Context Editing Syntax (by example)

1. B'CONTEXT' Copy to immediately before the string CONTEXT.
2. A!END! Copy to immediately after the string END
3. D/XYZ/ Delete the next occurrence of the string XYZ,
by copying to its start and skipping to end.
4. I/CHARS/ Insert the string CHARS into the output.

5.4.2.4 Other Commands

The following commands are not actually editing commands, but perform other useful functions.

The ‘WINDOW’ command (W)

The window command allows sections of the edited output to be printed on the monitor stream. It may have either of the forms:–

W integer e.g., W3
W

When used with no parameter the W command prints the current line on the monitor output stream. If the input pointer is at the start of the line, (i.e., the last character output was newline or newpage), the line is printed from the input stream. If on the other hand the input pointer is not at the start of the line, then the first part of the line is printed from the output stream, followed by the characters `~ ↑ ~` indicating the current position of the pointer, followed by the rest of the line from the input stream. Thus for example, if a line contained:-

BEGIN INTEGR I,J,K;

then the commands

D/INTEGR/I/INTEGER/W

would cause the following to be printed

BEGIN INTEGER~ ↑ ~ I,J,K;

The integer parameter N indicates that after printing the current line, the next N-1 lines of the input stream should also be printed.

The VIEW command (V)

The view command causes screen editing mode to be re-entered and the display will start from the current position onwards in the input stream to be made available for screen editing.

The RESTORE command (R)

This command simply restores the values of the input and output pointers to their values before the last positioning command, (i.e., S, C, A, B, D). This is useful if a command has been accidentally typed wrongly. For the purpose of this operation, the I command is not considered as an editing command.

The TOP command (T)

This command permits continuation of editing from the beginning of the edited files. It is equivalent to the commands.

```
E
ED outfile outfile.
```

Thus the edits done up to this point will appear in the new input file, and will have been secured on the disc.

The EXIT command (E)

This command copies the remaining characters in the input stream to the output stream and returns control to the program which called the editor (usually the job control interpreter).

If it is required to delete to the end of a file, this must be done by using the command SF.

The MERGE command (W)

This command has a filename as its parameter and causes the editor to switch its input to the start of the named file. Note that there is no way of returning to the previously selected file other than by a further Merge command, which will return to the start of the file.

The QUIT command (Q)

This command may be used to abandon an edit without updating the output file. It is also useful as an emergency exit in the case of disastrous errors.

5.4.2.5 Repetition of Commands

Any sequence of commands may be repeated a specified number of lines by enclosing the sequence in brackets () with a repetition count, e g.,

```
(10 S + 1 I/'COMMENT'/)  will insert the string 'COMMENT' at the
                        start of each of the next 10 lines.
(8 D/INTER/I/INTEGER/)  will correct a misspelling which has been
                        made 8 times.
```

Instead of the repetition count, any of the following may be used:—

- | | |
|---|--|
| L | Meaning repeat until a newline is encountered in the input stream. |
| P | Meaning repeat until a newpage is encountered in the input stream. |
| F | Meaning repeat until the end of the input stream is encountered |

When using the L and P repetition options, the pointers are left at the start of the next line or page. Thus for example:—

C+1. (P D 'E' I 'X')

will replace all 'E's on the next page by 'X's leaving the input and output pointers at the top of page after that.

Note that the only effect of the (L, (P options is to restrict the range of context searches to the current line or page respectively. Thus they may not operate correctly if searches for explicit line numbers are used within the repetition.

Examples

(L D/A/I/*/) Replaces all 'A's on the current line by '*'s.

(P D/AREA/I/AREA1/) Replaces all occurrences of the name AREA on the current page by AREA1.

(FA/MOD1/W) Searches for all occurrences of the name MOD1 in the file, and lists each line containing such an occurrence.

NOTE. In online mode the sequence of commands to be repeated is limited to a single line. There is no such restriction in offline mode. Brackets may be nested to a maximum depth of 5 though this should rarely be necessary.

Example

(5 (P D/A/I/B/)) Replaces all occurrences of the letter A by B on the next 5 pages.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 6

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

6 TERMINAL INPUT/OUTPUT

The procedures described in this Chapter are intended mainly for use by programs which interact closely with a user at a terminal. Owing to differences in the characteristics of different terminals, the precise character sequences which have to be sent to a terminal to perform a specific function, such as positioning the cursor or clearing the screen, are dependent on the type of terminal. To avoid knowledge of these differences propagating throughout all the software which needs to interact with the user, the system provides a set of procedures for performing input and output operations with a terminal such that the device dependencies do not need to be known by the application software.

The procedures fall into two main categories. The first includes the very low level primitives, such as those which send character sequences to the device to perform a specific function, for example clearing the screen. These are intended mainly for packages which require intimate control of the display, for example screen editors. The second set of operations provides a higher level interface With the user at the terminal via questionnaires and menus

6.1 LOW LEVEL TERMINAL OPERATIONS

1) VDU.NUMBER () / INTEGER

This procedure returns the device number of the primary terminal connected to a process. The device number may subsequently be used in commands, such as `CHANGE.CHANNEL` (see Section [15.2](#)) to alter the device characteristics.

2) VDU.TYPE () / INTEGER

This procedure returns an integer to identify the type of terminal connected to a process. It is divided into two 4-bit fields giving character display type and graphic display type. The meaning of the fields are as follows:

Character Display Types

- 0 ANSI (e.g. Zenith)
- 1 Univac UTS10
- 2 Newbury
- 3 Intertube
- 4 ICL
- 5 TTY

Graphic Display Types

- 0 No graphics capability
- 1 VME/10
- 2 Tektronics
- 3 Primagraphics Frame Store
- 4 Atari ST (remote)
- 5 Atari ST (local)

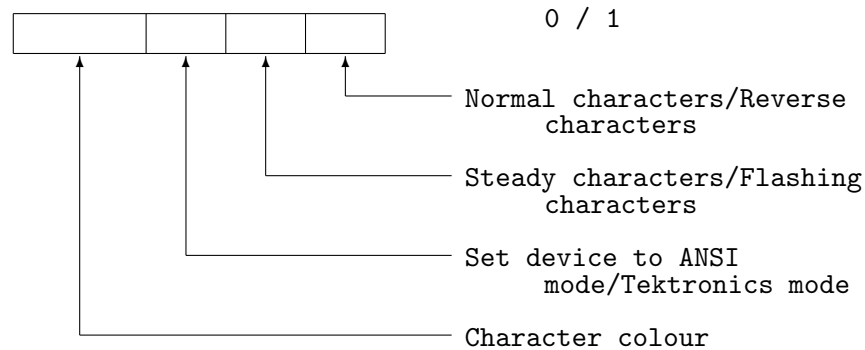
3) VDU.SIZE () / INTEGER32

The size of a VDU screen is usually expressed in terms of the number of lines on the screen (the rows) and the number of characters on each line (the columns). The result of this procedure has the number of rows in the most significant 16 bits and the number of columns in the least significant 16 bits.

4) SET.VDU.MODE (LOGICAL16)

This procedure defines the format of characters on the display and, where devices are capable of operating in more than one mode (e.g. character/graphics), enables the mode of the device to be switched.

The parameter is interpreted as two 8 bit fields; the most significant eight bits identify which characteristics have to be changed and the least significant eight bits indicate the nature of the change. Thus, according to the polarity of bits in the upper field. device characteristics may be either left unchanged (0) or modified (1) with the lower eight bits identifying the new state. The significance of bits in the lower byte is as follows:



5) VDU.MODE () / LOGICAL

This procedure returns the current mode of the device in a bit significant format. as described for SET.VDU.MODE.

6) POSITION.CURSOR (INTEGER, INTEGER)

This procedure positions the cursor at the specified row (P1) and column (P2) position on the screen. The top lefthand position is identified as row 0, column 0.

7) CLEAR.SCREEN ()

This procedure clears the entire screen. leaving the cursor at the top lefthand position.

8) CLEAR.LINE ()

This procedure clears the line from the current cursor position, leaving the cursor position unchanged.

9) READ.CHAR () / INTEGER

This procedure reads a character from the terminal and identifies sequences which constitute cursor movement operations. For cursor movements, the result of the procedure is

- 1 cursor up
- 2 cursor down
- 3 cursor right

-4 cursor left

For all other characters, the bottom eight bits of the result identify the character typed.

6.2 QUESTIONNAIRE AND MENU FUNCTIONS

The following procedures are intended to facilitate the production of questionnaires and menus, and to allow them to be displayed, modified or have items selected from them. (As the facilities support both the modification of fields and the selection of prespecified items, the display is usually referred to as a QUENU, being a cross between a questionnaire and a menu).

- 1) PREP.QUENU (ADDR [LOGICAL8], ADDR [LOGICAL8],
ADDR [LOGICAL16], ADDR [LOGICAL8]) / INTEGER32

This procedure is designed to facilitate the preparation of a questionnaire or menu. It produces the data structures needed for subsequent display and editing in the format required by the EDIT.DISPLAY procedure described later.

The parameter P1 identifies the basic proforma of the quenu. Three basic types of fields can be identified:

text which is simply intended to be informative to the user

menu items which may be selected by the user but which cannot be altered

questionnaire items which may be preloaded with suitable responses, may be altered by the user and subsequently selected.

Within the text string P1, menu items are delimited by ' and ' respectively, whereas the alterable questionnaire items are delimited by ' and ' respectively. Sequences of items of the same type may be inserted with the trailing delimiter omitted. Thus, for example, a sequence of menu choices might appear as

'choice 1'choice 2'choice 3'choice 4'

The delimiters are replaced within the final quenu of a single space.

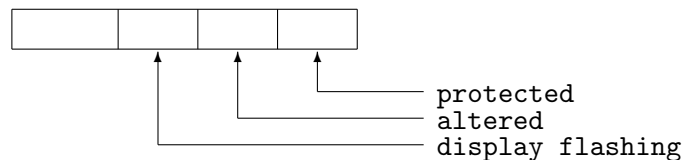
The output from the PREP.QUENU procedure appears in the vector parameters P2, P3 and P4 and as the result. P2 is a text string to be sent to the device, with all delimiter substitutions made. The size of the text string is returned in the most significant 16 bits of the result. The number of delimited (selectable) items is returned in the least significant 16 bits. The displacement within P2 of the selectable items in the quenu is returned in the vector P3. Thus, the least significant 16 bits of the result gives the number of items within this vector P4 returns additional information about each item it is made up of sequences of four bytes, holding respectively:

the row number of the item relative to the start of the quenu

the column number of the item, relative to the start of the line

the size of an item (in bytes)

status information about the item. The status information is bit significant, as follows



2) EDIT.DISPLAY (ADDR [LOGICAL8], ADDR [INTEGER16],
ADDR [LOGICAL8], INTEGER, INTEGER, INTEGER) / INTEGER

This procedure displays the text string P1 on the terminal and allows Items to be selected and modified. The parameters P2 and P3 identify the position of selectable items within the text string and additional status information about the items, as described in PREP.QUENU. Parameter P4 gives the position of the first line of the quenu on the screen. Thus, the entire quenu may be displaced relative to the top of the screen. P5 identifies the item upon which the cursor is to be positioned prior to editing. P6 identifies the device mode. If zero, the text will be displayed as normal characters, with the item at the cursor position highlighted in reverse video. If $P6 = 1$, reverse video is used for the text with the cursor position being shown in normal video.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 7

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

7 LIBRARIES

It is clear from the description of MUSS that its operation is very dependent on the availability, to every virtual machine, of a substantial collection of procedures in executable binary form. These procedures are called the system library. Facilities are also provided for the user to augment the system library with private libraries of executable code. This Chapter is concerned with the structure of these libraries and with the commands that allow compilers to compile calls on library procedures and command language processors to implement interpretive calls.

A similar mechanism applies to both libraries and programs, and of the procedures described below apply to both. The difference that libraries can be entered at one of many entry points by means of a procedure call which passes parameters, and a program has one entry point and no parameters are passed.

7.1 LIBRARY SEGMENTS

The compiling system (MUTL) generates library segments. These may be loadable or executable binary. This chapter is concerned only with the latter. It may be relevant to note that the binding of calls on other library procedures occurs when executable binary is created, either directly by a compiler or by a loader.

A library consists of a directory, the associated procedure bodies and possibly data segments. In some implementations these might occupy separate files but the file name of the directory is always regarded as the name of the library. A user need not be conscious of the additional files which constitute a library except that they will appear in the file store, with names derived by addition of pairs of characters (e.g. 00, 01 02) to the name of the library. A complete library, whether it be one or more files, contains the following

- a list of names for the procedures it contains,
 - their parameter specifications,
- a list of the names of the user defined
 - types used as parameters,
 - their specifications,
- a procedure entry table,

initialisation code,
initialised data structures,
and the procedure bodies.

The detail structure is machine dependent and is only relevant to the procedures described in this Chapter. However, the following is always true.

The library segment can be recompiled and, providing it contains the same procedures in the same order with unchanged specification, previously bound calls will continue to operate. This is still true even if new procedures specifications are added at the end. (In this instance, the ordering of the procedures is, in fact, the order of the procedure specifications (the LSPECS) for the exported library procedures, rather than the order of the procedures themselves).

7.2 THE SYSTEM LIBRARY

At the time a MUSS system is generated, a selection of library segments are included as the system library.

The content of this library can be different at different installations, but the commands described in this Manual will normally be provided, and extensions beyond this set are usually implemented as private libraries.

7.3 PRIVATE LIBRARIES

The commands `LIBRARY` and `RELEASE.LIB` were described in Chapter 2. The first dynamically extends the set of library procedures available, and the second reduces it by removing those of the named library.

A library can contain code at the outer module level, and this is treated as initialisation code to be executed at the time of opening a library by means of `LIBRARY`. Typically this code might create working segments and initialise data structures.

Sometimes it is necessary to discover the content of a library. The following command is provided for this purpose

LIST LIB (ADDR [LOGICAL8], ADDR [LOGICAL8], INTEGER)

The parameter P1 specifies a library name which must already have been opened (by LIBRARY). P3 indicates the level of detail required as follows

P3 = 0 procedure names only

P3 $\neq$ 0 gives procedure names and parameter types

P2 may be used to select a subset of the procedures. If P2 is non-zero only procedures whose names contain the string P2 will be listed.

7.4 PROCEDURES FOR CONSTRUCTING LIBRARIES

The procedures defined in this section are not normally used directly. They are the means by which MUTL creates libraries and programs and provides enquiry procedures by which the runtime diagnostics acquire access to the diagnostic information of libraries and programs.

1) INIT.DIR (ADDR [LOGICAL8], INTEGER) / INTEGER

This procedure creates a new empty directory for a library which will have name P1. P2 specifies the maximum number of procedures which are to be added to the library. If P2 = 0 the procedure will assume that a program rather than a library is to be compiled. INIT.DIR returns the segment number of the directory, and this is used subsequently as a parameter of the procedures which add to the library.

2) GET.SEG (INTEGER) / INTEGER

This procedure is called by MUTL to choose a suitable segment number for a segment of a program or library. This procedure chooses the highest numbered available segment that is less than P1.

3) ADD.PROC (INTEGER, ADDR [LOGICAL8], ADDR [LOGICAL], ADDR [INTEGER32]) / INTEGER

This procedure allows the specification of a new library procedure to be added to the directory, specified by segment number P1. P2 gives the name of the procedure and P3 is a list of integers which give the MUTL encoding for each parameter. P4 is a list which gives the dimension for

vector parameters. It returns a number which relates to the position of the procedure in the library. After a call of INIT.DIR the first procedure number returned will be 1 and it will thereafter increase by one on each call of ADD.PROC. This number should be used instead of the MUTLN in type specifiers which are pointers to the procedure.

- 4) ADD.TYPE (INTEGER, ADDR [LOGICAL8], ADDR [LOGICAL], ADDR [INTEGER32]) / INTEGER

This procedure allows the specification of a user defined type to be added to the directory specified by P1. P2 gives the name of the type and P3 is a list of integers which give the types of its constituent components in MUTL encoding. For each entry in the list P3, which is a vector, a dimension should be given in the list of integers which comprise P4. The result is a number which relates to the position of the type definition inside the library. Like the procedure number, it will count up from 1 on successive calls, and it must be used instead of MUTLN's in type specifiers.

- 5) ADD.SEGMENT (INTEGER, INTEGER, LOGICAL32, LOGICAL, INTEGER32)

This procedure is used to pass into the directory structure information relating to the code and data segments of a compiled library. If the segment exists at the time of the call of ADD.SEGMENT, and it is to be filed, then

P1 specifies the directory segment
P2 gives a compile time segment number
P3 gives a run time byte address
P4 gives the required run time access
(as in OPEN.FILE)
P5 gives the segment size in bytes.

If the segment is to be created as a scratch segment when the library is opened, P1 and P3 should be set as above, but P2 will be zero and P5 will give the required segment size.

The first segment added to a library by this command is assumed to be the principal code segment containing initialisation code and the

procedure entry table.

6) CLOSE.DIR (INTEGER)

This procedure completes the construction of a library directory and files the directory segment, specified by P1, under the name given in the call to INIT.DIR.

A library directory is typically formed, during compilation, by a call to INIT DIR, several calls to ADD.PROC and ADD.SEGMENT, and possibly ADD.TYPE, and finally by calling CLOSE DIR. The ADD.PROC and ADD.TYPE calls occur as each procedure or type is encountered, and ADD.SEGMENT is called at the end of compilation.

7) MERGE.DIR (ADDR [LOGICAL8], ADDR [LOGICAL8], ADDR [LOGICAL8])

This procedure creates a new directory P3 by combining directories P1 and P2. The code segments of directories P1 and P2 remain separate. The purpose of this is that the merged directory can be treated as a single library.

8) FIND.RD (ADDR [LOGICAL8], ADDR [INTEGER]) INTEGER

This procedure returns a list of segments in P2 containing diagnostic information for the library P1. For the current program P1 is an empty string. The result gives the number of diagnostic segments found. If new diagnostic information is available since the previous call on FIND.RD or FIND.ALL.RDS the %8000 bit is also set in the result.

9) FIND.ALL.RDS (ADDR [INTEGER]) INTEGER

This procedure returns a list of segments in P1 containing diagnostic information for all libraries programs loaded. The result is set as in FIND.RD.

10) FIND.SEG.RD (INTEGER) INTEGER

This procedure returns the segment number containing the diagnostic information for which P1 is a valid control address. The %8000 is set in the result as in FIND.RD.

11) FIND.LIB.NAME (INTEGER) ADDR [LOGICAL8]

This procedure returns the library name associated with the diagnostic information in segment P1.

12) FIND.PROC (ADDR, ADDR [LOGICAL8], ADDR INTEGER, ADDR [LOGICAL8], ADDR INTEGER) INTEGER

This returns the library and procedure name associated with the control address P1. The names are placed in P2 and P4 respectively, and the name lengths in P3 and P5. The result gives the principal code segment number. A zero result means the procedure was not found. If P1 is not in a library procedure then results for the library procedure preceding it are returned.

7.5 PROCEDURES FOR IMPLEMENTING CALLS ON LIBRARY PROCEDURES

These procedures are only needed by compilers and command interpreters.

1) FINDN (ADDR [LOGICAL8], INTEGER) / LOGICAL32

This procedure searches the lists of names associated with both private and system libraries, for the name specified by P1. P2 indicates whether the name is the name of a procedure (0) or type (1). It returns an integer, called a 'library index'. The value 0 means that the name is not present in the library. Normally the 'library index' is only used as a parameter to other procedures such as FINDP, FINDF and TL.PL, and users do not need to know the encoding. In fact the encoding is machine dependent.

2) FINDP (LOGICAL32, INTEGER, INTEGER, ADDR INTEGER32) / INTEGER

This procedure returns type information about a specified procedure. P1 is a library index, as described above, optionally qualified by a procedure number P2, as returned by ADD.PROC. If P2 is negative the procedure whose number is encoded in P1 is assumed, otherwise the procedure number in P2 is substituted. P3 identifies what information is required. If P3 = 0, the number of parameters to the procedure is

returned. If P3 specifies a parameter number, the MUTL encoding for that parameter is returned. If P3 equals the number of parameters +1, the MUTL encoding for the type of the result is returned. P4 is only relevant with vector parameters, in which case their dimension will be assigned to P4.

- 3) FINDF (LOGICAL32, INTEGER, INTEGER, ADDR INTEGER32) /
INTEGER

This procedure is the means by which the MUTL types of the constituent fields of a type are discovered. P1 is a library index as returned by FINDN. P2 is an optional type number (as returned by ADD.TYPE) and P3 specifies a field number. As in FINDP, P2 will be ignored if negative, otherwise it will replace the number in P1. Also if P3 = 0 the number of fields will be returned. When P3 is set to the number of fields of the type + 1 the procedure will return the alignment requirement of the type in bytes, if P3 is set to the number of fields + 2 the size of the type in bytes is returned. P4 is only relevant with vector fields in which case their dimension will be assigned to P4.

- 4) FINDTD (LOGICAL32, INTEGER, ADDR (INTEGER32)) / INTE-
GER

This procedure yields the position, in bytes, of fields within a type. P1 specifies the library index of a type as returned by FINDN and the offsets are placed in the vector P3. P2 specifies a subtype as in FIND.P.

- 5) LOOK.UP.N (INTEGER, ADDR [LOGICAL8], ADDR [LOGICAL8])
/ LOGICAL32

The purpose of this procedure is to yield information about the procedures and types in the library specified by P2. In particular it yields the library index of the nth name as a result, where n is specified by P1. It also copies the name into P3, preceded by a byte which indicates whether it is a procedure or a type (ms bit = 0/1), and gives the length of the name (ls 7 bits). If the library does not have the specified nth procedure a zero result is returned.

6) LINK (ADDR [LOGICAL8], ADDR [T]) / INTEGER

This procedure provides for an elementary form of dynamic linking. P1 should contain a list of names separated by spaces or newlines. P2 must be the address of a vector of ADDR PROC elements. For example using the notation of MUSL (Chapter 9) T might be defined thus

PSPEC T ();

This procedure will use the FINDN procedure for each name in P1 and thereby obtain its address which will be placed in the corresponding position in P2. If all the names are found a zero result will be returned, otherwise -1 will be returned.

7.6 INTERPRETIVE CALLING OF LIBRARY PROCEDURES

Sometimes it is necessary to interpret user source as immediate calls on library procedures. This occurs for example, in PPC.CMD. Thus it is necessary to stack a link, stack each parameter and then enter the procedure. These operations are all too low a level to be performed by the library procedures, therefore built-in functions are provided in MUSL (see Section 9.6.5.3).

MUSS

USER MANUAL

VOLUME 1

CHAPTER 8

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

8 OPERATOR FACILITIES

The facilities described in this Chapter are concerned with bootstrapping and configuring the system, management of user accounts, performance measurement and other miscellaneous user commands.

8.1 BOOTSTRAPPING

Each implementation of MUSS contains a monitor which provides a number of low level diagnostic aids, and facilities to downline load a MUSS system from another computer or read it from exchangeable discs, but most importantly for the operator it is the means by which the MUSS system is started.

On systems which contain readily accessible ROMs, for example, the Motorola VME/10, the monitor is placed in ROM. In other cases, for example VAX11/750, the MUSS monitor is on disc and the native monitor is used to read it into memory. This means that the procedure for entering the MUSS monitor is machine dependent. However, once the monitor has been entered the differences are minor, and the monitor provides a standard menu which allows the operator to select a version of MUSS and boot it in.

Normally the menu will offer a choice of two versions of MUSS and the operator types 1 or 2 to select the required one. (Whilst the machine is running one version of MUSS the other may be modified for subsequent testing and use.)

Facilities also exist in the monitor for partial or complete disc dumps. The details of this depend upon the type of disc drive available, and hence are machine dependent. Further information is given for each machine in a document entitled Supplementary Operator Information.

8.2 CONFIGURATION

The operator is provided with facilities for changing the configuration of a machine from that initially compiled into the system. The changes may affect both the peripheral process configuration of a machine, its status within a network, and the detailed device configuration.

Two options are available to an operator for changing the configuration.

A set of commands, as described below, may be called to change the configuration whilst the system is running. These changes will be lost when the system is restarted. For changes of a more permanent nature, a text file may be created containing a set of configuration parameters. This file will be processed and the configuration amended accordingly on every system restart. The file, which should be called CONFIG, must exist under the SYSTEM username. Unfortunately incorrect changes to this file may render the system unusable, even to the extent that the file cannot be corrected. The method of recovery from this state is machine dependent and is described in the supplementary notes mentioned above.

8.2.1 Terminology

It is necessary to understand some of the terminology used in describing the organisation of input/output devices before attempting to alter the configuration of the system. First, each 'device' attached to the system has a 'peripheral channel number'. A device in this sense is a socket to which an actual device can be connected. For example the socket on a multiplexor into which a VDU might be connected. The number of available peripheral channels on a given system is compiled in and each will relate to a particular physical connection point. However, the logical and physical nature of the actual plugged in device can be specified through the CONFIG command described later. Another use of the peripheral channel number is in forming a 'SPN' for a peripheral process. This is necessary, for example, in the SET.PERIPHERAL command, which allows a pseudo process name to be assigned to devices, such as LPT.

Another configurable attribute of a peripheral channel is its 'protocol'. The device protocol relates to the rules which are applied in mapping between actual device input/output and MUSS messages. Thus, for example, there is a separate protocol for each COMMS discipline. Issue 12 MUSS contains two COMMS disciplines which are similar. The one labelled ISSUE 12 is the preferred discipline and the one labelled ISSUE 11 is compatible with previous issues. Perhaps more importantly Issue 12 of MUSS is structured so that other standard COMMS protocols may easily be added.

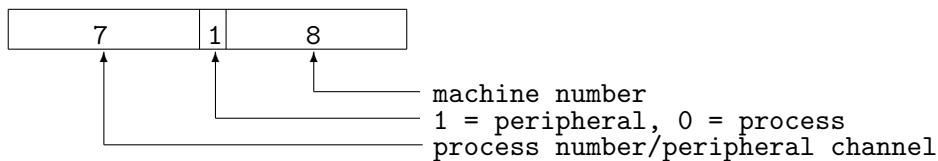
Protocols numbered up to 8 are standard on all MUSS systems, and those above 8 will vary between machines and implementations. The current allocation is as follows

```

Protocol 1 -- TERMINAL INPUT
Protocol 2 -- ISSUE 12 COMMS INPUT OUTPUT
Protocol 3 -- ISSUE 11 COMMS INPUT OUTPUT
Protocol 4 -- POSITIONAL DATA INPUT (TABLETS ETC)
Protocol 8 -- MU6 GRAPHICS INPUT OUTPUT (GENISCO)

```

A system process number (SPN) in general is formed from a process number and a machine number. However, one bit is reserved to indicate that the process number field is to be interpreted as a peripheral channel number thus:



The role of the peripheral channel was defined above.

8.2.2 Configuration Commands

When the system is in operation the communication between users and supervisors and user processes and peripherals is determined by the contents at the following tables

NETWORK TABLE –	this gives the name of each machine accessible from the current machine, and the number of the peripheral channel through which it can be accessed.
SUPERVISOR TABLE –	this gives for each supervisor in the network – a name, a machine name, a SPN, a PID and a channel.
PERIPHERAL TABLE –	this gives a name and a SPN for each peripheral process.

The first three commands below define the entries in these tables

1) SET.NETWORK (INTEGER, LOGICAL64, INTEGER)

This procedure changes entries in the NETWORK TABLE. The first

parameter is the table entry to be changed. P2 specifies a machine name, and P3 the peripheral channel number which will be used for communication between the current and specified machine.

2) SET.SUPERVISOR (INTEGER, LOGICAL64, LOGICAL64, INTEGER, INTEGER, INTEGER)

This procedure changes entries in the SUPERVISOR TABLE. The first parameter is the table entry to be changed. P2 and P3 specify a process name and machine name respectively for the supervisor process and P4, P5 and P6 are the SPN, PID and channel for communication with the process.

3) SET.PERIPHERAL (INTEGER, LOGICAL64, INTEGER)

This procedure changes entries in the PERIPHERAL TABLE. The first parameter is the table entry to be changed. P2 specifies a peripheral name and P3 the SPN to be associated with it. If P4 is zero, processes sending messages to the peripheral will not be suspended, if non-zero they may be suspended.

4) CONFIG (INTEGER, LOGICAL, LOGICAL, LOGICAL32, INTEGER)

This procedure is used to change the configuration of devices in the system.

Its parameters are

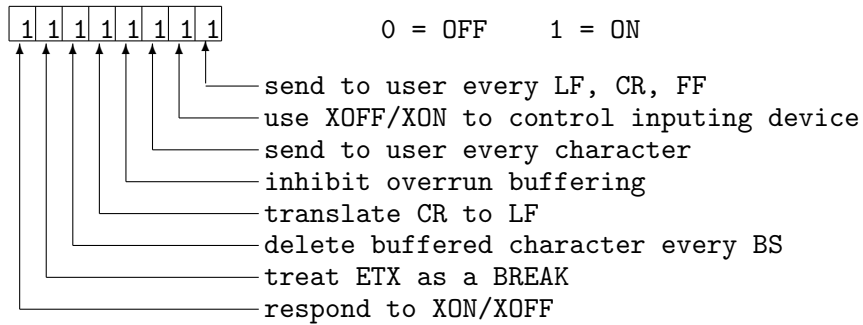
```
P1 -- Channel number
P2 -- Protocol (as described in 8.2.1)
P3 -- Protocol mode
P4 -- Physical line parameters
P5 -- Terminal type.
```

P1 specifies a peripheral channel number

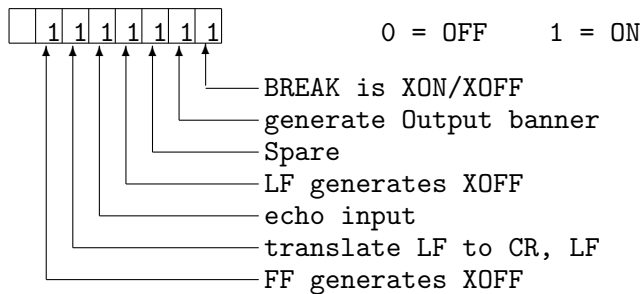
P2 specifies the protocol number

P3 gives information which steers the actions of the protocol. Its least significant 6-bits relate to special output actions and the next 8-bits relate to special input actions. For the terminal I/O protocol these are as follows

Input:

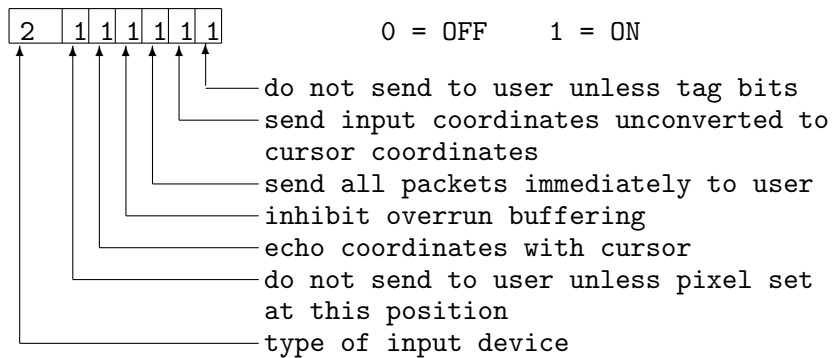


Output:



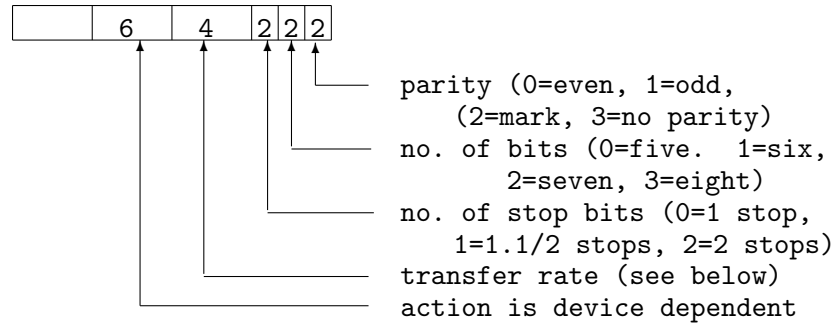
For the positional data input protocol the output is as for the terminal protocol while the input bits are as follows

Input:



For other protocols all bits must be zero.

P4 gives line parameters which specify the physical characteristics of the device as follows



The following encoding may be used to specify different transfer rates:

<u>code</u>	<u>baud</u>	<u>character/s (approx.)</u>
0000	110	10
0001	300	30
0010	1200	120
0011	2400	240
0100	4800	480
0101	9600	1K
0110	19200	2K
0111	38400	4K
1000	76800	8K
1001	153600	16K
1010	307200	32K
1011	614400	64K
1100	1228800	128K
1101	2457600	256K
1110	4915200	512K
1111	9830400	1M

P5 represents the default type of terminal expected on the line. Its value is available to user and library software through READ.IO.STATUS.

The maximum number of devices and peripheral channels which may be configured are determined at compile time and P1 and P2 should be kept within these limits.

5) SET.COMMON.SEGMENT (ADDR [LOGICAL8], LOGICAL64, INTEGER, LOGICAL)

This procedure causes a specified file to become a common segment. P1 and P2 give the filename and username of the file while P3 and P4 give the common segment number and its required access

6) READ.IO.STATUS (INTEGER)

This procedure gives information on the I/O channel with SPN P1. It returns the following information in the PW variables:—

```
PW1 -- Terminal type
PW2 -- SPN of the current connection
PW3 -- SPN of the primary connection
PW4 -- Protocol mode
PW5 -- Protocol number
PW6 -- Physical parameters
```

8.2.3 Configuration File

The CONFIG file consists of a set of keywords identifying the type of parameter to change, followed by the changes. The keywords are preceded by @. The following options are available

@MACHINE	This should be followed by an integer specifying the number of the machine within the network.
@NETWORK	This allows the operator to change the configuration of the machines known to the system. The keyword is followed by a list of machine names and associated peripheral SPNs. These correspond to the parameters for the SET.NETWORK command.

@SUPERVISOR	This allows the operator to change the list of supervisor processes known to the system. The keyword is followed by a list corresponding to the parameters of the SET.SUPERVISOR command, of process and machine names for the known supervisors, and their associated SPN, PID and channels.
@PERIPHERAL	This allows the operator to change the list of peripheral names wired into the system. The keyword is followed by a list corresponding to the parameters of SET.PERIPHERAL, of peripheral names and their associated peripheral SPN's.
@DEVICE	This allows the operator to set/change device configuration. It has a parameter list which corresponds to that of the CONFIG command.
@COMMON	This allows files to become common segments in the systems virtual store. It has a parameter list which corresponds to that of SET.COMMON.SEGMENT.
@TERMINAL	This allows terminals to be automatically connected to processes. The parameters are the terminal channel number and the process SPN.
@END	End of parameters.

All numbers appearing in the file must be hexadecimal.

A typical CONFIG file for configuring a machine might be:

```
@MACHINE
3          This is machine number 3 in the network.

@NETWORK
MU6G  2
VAX02 2
PDP44 2
VME   0
```

Four other machines are accessible although they are all apparently connected to the same channel which implies that only one is actually connected and the others are connected to it.

@PERIPHERALS

OPR 303
LPT 303
LPTVX02 4506

Three peripheral processes are known, with the first two associated with the same device.

@SUPERVISORS

JOB VAX02 206 1 8
FILMAN VAX02 406 2 8
FILMAN MU6G 0400 2 8

These entries allow access to the JOB process of VAX02, and the filestores of VAX02 and MU6G.

@DEVICES

1 1 F130 0148 This is a 9.6 Kilobaud character line.
2 2 0000 014C This is a 9.6 Kilobaud 'New' Comms line.

@COMMON SEGMENTS

BFI0500 UTIL 3A D
BPI0500 UTIL 3B D
BBI0500 UTIL 30 D
BCI0500 UTIL 31 D
@END

8.3 ACCOUNTING AND USER MANAGEMENT

Controlling users and accounting for their use of the system is dependent on conditions in a particular installation. The aim of the MUSS accounting system is to provide a set of primitive operations that the System Manager of an installation can use to build an accounting regime appropriate to his needs.

The basic system maintains accounts for four resources. These are:

- | | |
|------------------|---|
| (i) CPUTIME | The amount of CPU time used (in seconds). |
| (ii) NO OF FILES | The maximum number of files that a user can own. |
| (iii) FILESTORE | The maximum amount of filestore (Kbytes) a user can occupy. |
| (iv) SUBORDS | The maximum number of subordinates a user can have. |

The user SYSTEM has infinite amounts of all resources which it can distribute to other users in the system. Any other user can create new subordinate users, provided that the user has an allocation for SUBORDS.

8.3.1 Commands

The commands associated with accounting and user management fall into two categories, basic (primitive) commands and high-level commands. The primitive commands are those which deal with separate accounting and user management tasks, while the latter commands provide a more convenient interface for some typical user management activities.

8.3.2 Primitive Commands

The set of consumable balances and incomes and reusable maxima for a user is called his 'parameters'. The parameter numbers for the standard resources are:

CPU time balance	- 1
CPU time income	- 3
Maximum files	- 4
Maximum file store	- 5
Maximum subords	- 6

1) CREATE.USER (LOGICAL64)

This command creates a new user with the name specified in P1 as a subordinate of the current user. The name must be unique within the installation.

2) ERASE.USER (LOGICAL64)

This procedure removes the subordinate user (with username P1) from the system. The subordinate's balances of consumables and maxima of reusables are added to the current user's corresponding values. The subordinate must not have any subordinates or have any reusables currently allocated.

3) SET.USER.PARAM (LOGICAL64, INTEGER, INTEGER)

This procedure re-distributes a balance, maximum, or income between a user and one of his subordinates (P1). P2 is the parameter number, P3 is the amount (POSITIVE) to which the parameter is to be set. The change must not reduce either user's parameters to less than zero.

4) NEW.PASSWORD (LOGICAL64, LOGICAL64)

Changes the password of the current user (P2 zero) of one of his subordinates (P2 is the username) to the value in P1.

5) CATALOGUE.USERS ()

Returns a new segment containing a table of information on the current user and his subordinates (if any). There is one entry for each subordinate. and a subordinate's subordinates immediately follow this entry (defined recursively). The format of each entry in the table is:

USERNAME (8 bytes)
PASSWORD (8 bytes)
LEVEL (1 byte) *
spare (3 bytes)
BALANCE 1 (4 bytes)
BALANCE m (4 bytes)
INCOME 1 (4 bytes)
INCOME m (4 bytes)
MAXIMUM 1 (4 bytes)
MAXIMUM n (4 bytes)

*This will be 0 for current user.
 1 for immediate subordinate.
 2 for subordinate's subordinate
 and so on.

PW1 holds the number of users in the table, PW2 holds the segment number. PW3 is set to the maximum number of subordinates that the user is allowed to create. PW4 holds the number of consumable resources and PW5 the number of reusable resources (the m and n values respectively in the figure above).

6) PAYOUT ()

This procedure causes each user's income for each consumable resource to be added to the appropriate balance and then, if necessary, this is reduced to the limit. It may only be invoked by the system manager.

7) FIND.U (LOGICAL64, LOGICAL64)

This procedure returns in PW1 the UID of the user specified by username (P1) and password (P2). The UID of a subordinate may be obtained without giving a password.

8.3.3 High-level Commands

1) NEW.USER (ADDR [LOGICAL8], ADDR [LOGICAL8])

This procedure creates a new set of users as subordinates of the current user.

Prior to the creation of individual users, values for new users resources are read from P1. This should contain CPU time income, maximum files, maximum file store (K byte blocks) and maximum subordinates. Entries are all unsigned decimal integers on separate lines. An empty line or one which has other than an integer value in it is interpreted as a zero. This list may be terminated using an '@' on a separate line, in which case the rest of the resources will be set to zero.

Names of users are read from another document (P2). This must contain pairs of username and password, each on a separate line. The default for password is 'X', however if more users are to be created the default password must also appear in P2. Operation may be terminated by an '@'.

Default for P1 and P2 is the currently selected stream. If the command is used without arguments in interaction mode, appropriate prompts will appear to steer the operation.

2) DELETE.USER (ADDR [LOGICAL8])

This procedure removes a user's subordinate from the system, transferring the resource allocations back to the superior. The subordinate must not have any subordinates or have used any other reusables. P1 is the source document which contains names of users to be deleted, terminated by an '@'. The default is the current stream.

3) SET.ACCOUNT (ADDR [LOGICAL8], ADDR [LOGICAL8], ADDR [LOGICAL8])

This command redistributes balance, income and maxima between a user and one or more of his subordinates. It works in a similar way to the command NEW USER by reading new income/maxima, new balance (CPU time) and finally usernames.

P1 is the source from which positive integers indicating the new values of the user's income/maxima are to be read.

P2 is the source from which balance is read. Both P1 and P2 may be terminated by an '@' on a separate line. In each case the remaining resources will be left intact.

Usernames will be read from P3, one name per line and terminated by an '@'. The default for P1, P2 and P3 is the currently selected stream.

4) LIST.ACCOUNTS (LOGICAL8, ADDR [LOGICAL8])

This command lists the accounts parameters for the current user and his immediate subordinates (P1 equals S), or all his subordinates (P1 equals A).

For each user, the name and level (0 = self, 1 = subordinate, etc.) and a set of accounts with appropriate captions are output to the destination document (P2). The currently selected output stream is the default for P2.

8.4 PERFORMANCE MEASUREMENT

This section describes the procedures provided in MUSS for interactive benchmarking and collecting statistics about the system.

8.4.1 Commands

There are two categories of commands associated with benchmarking and performance measurement, namely an organisational function for acquiring system statistics and a number of high-level procedures for printing statistics and benchmarking.

8.4.2 Organisational Commands

1) SYSTEM.STATS ()

This procedure returns (in PW1) a new segment, containing statistics about the system since restarting.

The following integers will be returned in the segment:

- Number of Organisational Commands
- Number of Seconds at System Level
- Number of Seconds at User Level
- Number of Seconds Idling
- Number of Cache Loads
- Number of Cache Resets
- Number of Drum Page Transfers In
- Number of Drum Page Transfers Out
- Number of Pages Allocated on Drum
- Number of Disc Sectors Transferred In
- Number of Disc Sectors Transferred Out
- Elapsed Time (Seconds)
- Number of Processes Created
- Number of Virtual Store Interrupts
- Number of Segments Created
- Number of X Segments Created
- Number of Pages Requested with the Page already in Memory
- Number of Invalid Memory Accesses
- Total number of Pages Rejected
- Number of Pages Rejected to Disc
- Number of Long Messages Sent
- Number of Short Messages Sent
- Number of Files Opened
- Number of Files Created
- Number of Files Updated
- Number of Tasks Set.

8.4.3 High-Level Procedures

Two procedures are provided for noting and printing the statistics collected about the system:

1) READ.STATS ()

This procedure notes the current values of the system statistics.

2) OUT.STATS (INTEGER)

This procedure prints the statistics collected. Two options are available. If the parameter is zero, the difference in statistics since the last call of READ.STATS will be printed. Thus a program can monitor the performance of the system between any two points.

If the parameter is non-zero, or if READ.STATS has not been called, the statistics printed relate to the behaviour of the system since restarting.

3) INT.JOB.JOB (ADDR [LOGICAL8], LOGICAL8)

This procedure implements an interactive benchmarking facility. It runs a set of interactive jobs, supplied as input ‘scripts’ on files. Each script is run from a specified number of terminals, and may, if required, be run more than once from each terminal. The scripts should not contain the command STOP.

Input is supplied to the jobs a line at a time, each line being sent only after receiving a prompt in response to the preceding line.

On completion of all scripts, the system statistics collected during the session are output to a log. In addition a complete record of all short messages input and output is kept in the log.

P1 is the name of the output document (log) on which the session log is to be output. If P1 is not specified, the currently selected output stream will be used. P2 is a numeric delimiter character used to parameterise the scripts. If this is non-zero, then any occurrence of this character within the scripts will be replaced by a numeric terminal identifier. As a result several terminals may perform similar scripts operating upon different files.

Delays can be put in the scripts by inserting lines which start with ‘~’ followed by either an integer, specifying the delay in seconds, or by the delimiter character, which generates random delays.

A list of the files containing the scripts, with the number of terminals and

number of repetitions for each script, should appear on the current input stream, terminated by the character '@'.

Example:

```
INT.JOB.JOB(JOB,LOG.FILE,?)
SCRIPT1 20 5
SCRIPT2 5 15
@
```

If INT.JOB.JOB is initiated interactively, appropriate prompts will steer its use.

8.5 MISCELLANEOUS COMMANDS

1) CONNECT (INTEGER, LOGICAL8)

This procedure is intended mainly for use on small MUSS systems which might be used as 'terminals' on an alien system. The parameter P1 specifies a system process number, which should refer to a peripheral channel, and the assumption is that this channel is connected to an interactive port of the alien system. Control is retained by the CONNECT command, subsequent input is routed to the alien system and its responses appear on the user terminal. In addition the CONNECT command implements a primitive control language which facilitates file transfer and downline loading. The command lines begin with the warning character specified by P2 ('*' is the default) and such lines are not transmitted to the alien system. The commands are

- *L filename – this command creates an output stream, with destination = the given filename, and copies all subsequent input received from the alien system to this stream in addition to sending it to the users terminal.
- *S – this command stops the copying of input into the stream created by *L.
- *C – this command causes copying to be resumed.
- *T filename – this command transmits the specified file to the alien system, a line at a time. It waits between lines for a message to be returned which it assumes to be a prompt. All input received during transmission of the file will appear on the terminal.

*E – this command causes the CONNECT procedure to return control to command level. Any stream set up by the *L command will be broken at this time.

2) TEMPORARY.LABEL (INTEGER, LOGICAL64. ADDR ([LOGICAL8])

This command allows an operator to associate a new name and label with the tape on a tape drive or an exchangeable disc. P1 specifies the tape unit or disc drive unit number P2 and P3 give the name and label of the tape/disc respectively. Subsequent requests to mount a tape or disc (see Chapter 17) will recognise the medium by this new name and label, rather than by any label on the medium itself.

This command should be called prior to mounting the tape on the unit or inserting a disc in the disc drive.

3) CANT.DO (LOGICAL64)

This command may be called by the operator to refuse the request by a process for resources (such as the mounting of a tape or exchangeable disc) The parameter gives the name of the process.

4) SET.TIME.AND.DATE (LOGICAL64, LOGICAL64)

This command should be called by the operator immediately on starting up the system. The first parameter is the time of the day, consisting of the six characters for HRS HRS MINS MINS SECS SECS. The second parameter is the date, again expressed by six characters, DAY DAY MONTH MONTH YEAR YEAR.

The operator should check the validity of the time and date after calling this command, using the procedures OUT.TIME and OUT.DATE (see Chapter 4).

5) TIME.AND.DATE ()

This procedure returns in PWW1 the current time and date, expressed as the number of seconds since midnight on 1st January, 1900. The number of milliseconds since the system was last restarted is returned in PWW2.

6) LIST.JOBS ()

This command allows the operator to list all the processes currently in the system, along with useful information concerning each process. The status of the process is indicated as follows

- M – waiting for a message
- S – suspended (by suspend command)
- T – suspended awaiting termination
- I – suspended for direct I/O
- P – suspended by semaphore operation

7) LABEL.DISCS (INTEGER)

This procedure is used to write a name and label to drive P1. Each disc should be placed in the drive and then the required name and label input in response to the prompt. The character '@' in response to a disc name request terminates the procedure.

8) READ.MACHINE.STATUS ()

This procedure returns information about the machine and system currently running. PWW1 is the machine name and PW1 is the number of the machine in the network. PW2 is the version number of the operating system.

9) DESPOOL (ADDR [LOGICAL8], ADDR [LOGICAL8], INTEGER)

This procedure outputs all the files in a given directory to a given peripheral. P1 is the directory name where the files are to be found. P2 is the process name of the peripheral to which the files are to be output (e.g LPT*, HPP*). P3 determines the mode in which the procedure operates. If it is zero each file will be deleted from the directory when its output is complete. Also the procedure will never return but will periodically check for any new files to be output. However, if P3 is one, no files will be deleted and the procedure will return once all the files present at the time it is called are output.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 9

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

9 SYSTEM PROGRAMMING LANGUAGE

9.1 INTRODUCTION

The implementation programming language of MUSS is the Manchester University System Programming Language (MUSL). Its overall structure reflects its principle function, namely the implementation of MUSS. In designing such a language for operating system implementation one of two approaches can be taken. First an attempt can be made to formally cater for the operating system requirements in the 'virtual machine' provided by the language. For example, the notion of tasks (or processes) and multi-tasking can be included. However, this tends to move some of the operating system functions into the runtime package of the language rather than providing the means for their implementation. The second approach, which is the one followed in MUSL, is to allow direct access, through the language, to the actual machine. Thus an operating system written in MUSL controls directly the use of store, CPU and peripherals. Many of the special features of MUSL will not be required for writing simple user programs, and the underlying basic language is straightforward and Pascal-like.

For the purpose of describing the syntax of MUSL this manual uses a variant of BNF. This will be mostly self evident to readers familiar with this kind of notation. For example

$$\langle A \rangle ::= W!X[Y!Z]$$

means that the syntactic element A may be W, XY or XZ. Also

$$\langle B \rangle ::= \langle A \rangle [\langle B \rangle ! \langle \text{NIL} \rangle]$$

means that B may be

W or XY or XZ
or WW or WXY or WXZ
or XYXY or XYW or YXXZ
or XZXZ or XZW or XZXY
or WWW or WWXY or WWXZ
and so on

NIL is the only reserved name used, but some notation is required for the symbols $\langle, \rangle, !, [,]$ and this is $\langle \langle \rangle, \langle \rangle \rangle, \langle ! \rangle, \langle [\rangle, \langle] \rangle$.

9.1.1 Modular Structure

Software written in MUSL is normally subdivided into modules. A single module is all that the compiler will accept. In the simplest case a module can be a complete program, but more commonly a program will consist of several modules and MUSS consists of many modules. Since the MUSL compiler only deals with single modules, it follows that the compilation of multi-module software will involve several calls on the MUSL compiler (one for each module), and that provision for inter-module linking has to be made outside of the compiler. This Chapter is mainly concerned with the rules for writing single modules but some discussion of module linking is appropriate to set the context.

9.1.2 Module Linking

Like all other compilers in the MUSS system the MUSL compiler generates code indirectly through calls on the MUTL (target language) procedures. These procedures can be instructed to generate either relocatable or executable binary. In the former case inter-module linking is postponed until the final link-loading pass, whereas in the latter case it becomes the responsibility of the MUTL procedures, and is carried out as the compilation proceeds. The MUTL procedures and the link-loading scheme are described elsewhere (See Section 7.5) but to give an understanding of how modules of MUSL fit together some repetition is appropriate here. It will be sufficient to consider the case where the MUTL procedures are generating executable binary, and hence performing the inter-module linking.

9.1.3 The Storage Model

Statements described under Subsection 9.4.1 allow a user to separate both the code and the data, contained in a module, into *areas*. This facility might be used in a module of the operating system, for example, to separate the code that needs to be resident in main store from that which might be paged, and similarly for the data structures. The MUTL system provides the means for areas to be placed at actual addresses.

MUTL considers the actual store to be subdivided into *segments*. Each of these has a number, for identification purposes, a size, a runtime address and a compile time address (all in units of bytes). They are defined by calling the MUTL procedure TL.SEG. In fact TL.SEG has one further parameter as described in Chapter 18, but its function is not appropriate to understanding

MUSL. Areas are related to segments by the MUTL procedure

TL.LOAD (MUTL segment number, MUSL area number).

Area 0 has a special significance of which the user must be aware. It is the runtime procedure stack (see Section 9.2.5), whose placement is determined by the MUTL system rather than by the user.

After the storage requirements have been specified for the set of modules that require to be linked, they are compiled by a sequence of calls on the MUSL compiler one for each module. Unless the defaults described in 9.1.6 are adequate each module should be preceded by calls on TL.LOAD to cause proper placement of its areas. As the compilation proceeds, the code and data items of the areas will accumulate in their respective segments. If more than one area of a module is mapped into a given segment, some interleaving may occur depending upon the placement of area selection directives (see Section 9.4.1) in the source.

9.1.4 Initialisation

It will be obvious from what has gone before that the MUTL procedures will require data structures that carry information across many individual MUTL procedure calls. These data structures must be initialised. Therefore at the start of a compilation the procedure TL (MODE) should be called. Also there are some final checks required after the last module of a compile job and the procedure TLEND must be called.

9.1.5 Modes of Compilation

For a 'normal' compilation of a user program the value 0 for the MODE parameter of TL selects appropriate options. The full range of options are described in Section 2.5 and in Section 18.11, Item 6.

9.1.6 The Command Structure of a Typical Compile Job

In the most general case the sequence of commands required to control a compilation would be

- a call on TL
- one call on TL.SEG for each segment
- one call on TL.LOAD for each area of the 1st module

a call on MUSL to compile the 1st module
a further sequence of calls on TL.LOAD followed
by a call on MUSL for each additional module
finally a call on TL.END.

However if the MUSL compiler is called with appropriate zeros in its MODE parameter it will call TL and TL.MODULE at the start and TL.END.MODULE and TL.END at the end. The least significant 8 bits of the compiler mode will be used as the TL mode.

When TL is called with a 'normal' MODE, segment 0 is automatically created (for code) and area 1 is automatically mapped into it at the start of each module. As stated earlier, area 0 is the procedure stack, and its placement in store is controlled by MUTL. That is area 0 cannot be mapped into a user created segment. Thus, for a simple program TL.SEG and TL.LOAD calls are not necessary.

Also unless directives (see Subsection 9.4.1) are used to control the placement of code and variables, all code will be placed in area 1 (segment 0) and variables will be placed in area 0 (the stack). In more complicated situations segments are created at the start of a compilation, and the mapping of areas into segments is given separately for each module.

9.1.7 The Format of a Module

In the general case a module will use procedures, and possibly literals, data structures, types and labels declared in other modules. Declarations for these entities, known as the 'imports' to the module, must precede the module heading. The module heading itself serves three functions. First it indicates the start of the module and the entities declared after the heading are private to the module unless they are formally 'exported'. The second function is to allow these exports to be specified as a list of names enclosed in parentheses. The third function is to provide the option of assigning a name to the module. If this option is used then all references to the exports from the module in other modules must prefix the name of the exported entity by the module name. A module is terminated by '*END' and on encountering this the compiler will 'exit'.

More formally the syntax of a module is

```

<MODULE> ::= <IMPORTS>
           MODULE[<NAME>!<NIL> ]<EXPORTS>
           <STATEMENTS>
           *END

```

The IMPORTS are declarations of entities exported from other modules. Their exact form will be discussed in Subsection 9.5.11, however, it should be noted that they are of two kinds. If the full detail of an imported entity is needed by the compiler, in order to compile references to it inside the module, a full duplicate declaration must be used. If there is no relevant detail a special import declaration (see Section 9.5.11) is used that gives only the name and kind of the entity.

The EXPORTS of a module are an optional list of names of entities declared within the module, thus

```

<EXPORTS> ::= (<NAME.LIST>)!<NIL>
<NAME.LIST> ::= <NAME>[,<NAME.LIST>!<NIL>]

```

For example, a module M2 that imports a procedure P1, which is exported from another module M1, and exports itself two procedures P2, P3 would have the general form

```

PSPEC M1.P1(INTEGER)
MODULE M2(P2,P3)
PSPEC P2(INTEGER)
PSPEC P3(INTEGER)
PROC P2(I)
.
.
statements of P2
.
.
END
PROC P3(J)
.
.
statements of P3
.
.

```

```
END
*END
```

It is assumed in the example that P1, P2, P3 each have one integer parameter. Obviously any external references to P2 and P3 must use the names M2.P2, M2.P3.

STATEMENTS are discussed more fully in subsequent Sections but it might be helpful to give an approximate picture of how they relate to modules and programs at this point. They fall into two main groups, declaratives and imperatives. The norm is for the declaratives to come first. Again the norm will be for most of the statements in a module to be concerned with defining procedures. That is, they will be contained within procedure bodies that are delimited by the procedure headings and the matching ENDS. Thus a module will typically take the form

```
import declarations
module heading
declaratives
imperatives
    procedure heading
    procedure body
    END
    .
    .
    .
*END
```

9.1.8 Kinds of Modules

There are now several cases to consider in order to convey the significance of modules.

If the module is an ordinary program, a run of the program implies executing the imperative statements, in the main body of the module. These may call the procedures enclosed in the module which will cause the imperatives that they contain to be executed. Thus the imperatives at the outer level might be regarded as the ‘main program’. A simple variant of this case is where the module is one of several that make up a complete program. In this case the imperative parts of each module will be joined together in the order

in which they are loaded. In most of the modules the imperative statements, in their main body, will usually be concerned with initialising the global data structures of the module, whilst that of one module will form the main program. Clearly the user must be conscious of the order of execution.

Sometimes in a program that consists of several modules, some of the modules may be entirely passive, in that they contain no imperatives in their main body. In this case the only way that the procedures of the module can be executed is through calls of the exported ones occurring in other ‘active’ modules.

Modules of this kind can be treated as libraries, which are compiled and filed, to be repeatedly used by other programs after being ‘opened’ by use of the LIB command. The main body of a library module may in fact contain imperatives statements, in which case they will be executed when the library is opened. Their function, therefore, is to initialise the data structures of the module.

Some modules form part of the MUSS system library. They are permanently open, hence they cannot have initialisation statements. If initialisation is required, as in the case of the MUTL procedures, a procedure (e.g. TL) must be provided that can be explicitly called by the user.

9.2 KINDS OF STATEMENTS AND SCOPE RULES

9.2.1 Kinds of Statement

There are a number of places in the language specification where an arbitrary sequence of statements occur. Thus there is a need for the definition

$$\langle \text{STATEMENT} \rangle ::= \langle \text{STATEMENT} \rangle [\langle \text{STATEMENTS} \rangle ! \langle \text{NIL} \rangle]$$

This section is concerned with various kinds of STATEMENT that exist in the language. It will be apparent that not all kinds of statement are appropriate in every context and norms will be suggested. However, the detailed constraints will only be given as the individual statements are described in later chapters.

There are five main kinds of STATEMENT as follows

$$\langle \text{STATEMENT} \rangle ::= \langle \text{DECLARATIVE.STATEMENT} \rangle !$$

```
<IMPERATIVE.STATEMENT>!  
<DIRECTIVE.STATEMENT>!  
<PROC.DEFN>!  
<BLOCK>!
```

9.2.2 Declaratives

The declarative statements normally appear at the start of a MODULE, PROCEDURE or BLOCK. With the single exception of LABELS (see Section 9.5) a declaration must be given for every name introduced into a module, before any other use of the name is permitted. The main function of declarative statements is to create entries on the name and property lists of the compiler. They do not, directly, result in the generation of executable code although they may contribute to the code, executed at procedure entry and exit time, that is concerned with dynamic storage allocation.

9.2.3 Imperatives

These statements are the means by which the computations are specified. Normally each MODULE, PROCEDURE and BLOCK will have a sequence of imperative statements following its declarative statements. They are fully discussed in Section 9.7, but briefly they consist of the usual forms of statement such as:

```
statements that express computations  
procedure calls  
FOR/WHILE loops  
IF/THEN/ELSE statements  
switches
```

9.2.4 Directives

These statements relate more directly to the compiler and the underlying MUTL than to the language. Their function is to select ‘modes’ of compilation. Some directives merely set switches in the compiler, whilst others allow the environment of the compilation to be manipulated by means of direct calls on the MUTL procedures and other MUSS Library Procedures. Obviously the positions in which they are placed must be carefully chosen in relation to the action they have. Some guidance on this is given along with their descriptions in Section 9.4.

9.2.5 Procedure Definitions

A procedure has to be both declared and defined. Procedure declarations, which are described in Section 9.5, specify the name of a procedure, and the number and type of its parameters and result. A procedure definition gives the code body of the procedure that is to be executed on each call.

A procedure is defined by a heading followed by a sequence of statements terminated by END.

$$\begin{aligned}\langle \text{PROC.DEFN} \rangle &::= \langle \text{PROC.HEADING} \rangle \\ &\quad \langle \text{STATEMENTS} \rangle \\ &\quad \text{END}\end{aligned}$$

Normally the STATEMENTS will consist of DECLARATIVES followed by IMPERATIVES.

The function of the PROC.HEADING is to give the name of the procedure and the names of any parameters that it may have.

$$\langle \text{PROC.HEADING} \rangle ::= \text{PROC} \langle \text{NAME} \rangle [(\langle \text{NAME.LIST} \rangle) ! \langle \text{NIL} \rangle]$$

At runtime each call of a procedure causes space to be allocated dynamically on a ‘procedure stack’ containing linkage information, parameters, and the variables declared within the procedure. Hence procedures may be recursive.

Although every procedure must be declared before it can be used or defined, the definition need not precede calls on the procedure. The only restrictions on the placement of the definition of a procedure are that it must come after the declaration of the procedure and be at the same level of the block structure (see Subsection 9.2.7).

9.2.6 Blocks

A BLOCK is a sequence of statements delimited by BEGIN and END.

$$\begin{aligned}\langle \text{BLOCK} \rangle &:= \text{BEGIN} \\ &\quad \langle \text{STATEMENTS} \rangle \\ &\quad \text{END}\end{aligned}$$

Normally the statements will consist of IMPERATIVES possibly preceded by some DECLARATIVES. The main purpose of the BLOCK construct is to represent a group of statements as a single statement. This is useful in some of the control constructs described in Section 9.7 where only a single STATEMENT is admissible. Another use of the BLOCK construct is to localise the ‘scope’ of the entities declared in the block to the statements that it contains.

9.2.7 Scope and Block Structure

The language allows blocks and procedures to be nested inside each other to any depth and the implications of this must be understood. Consider the module shown schematically below

```
MODULE M
  declaration of procedure A
  declaration of procedure B
  .
  .
  PROC A
    declaration of procedure C
    .
    .
    PROC C
      .
      .
    END
  END
  PROC B
    .
    .
  END
*END
```

Subject to certain restrictions the overall scope rule is that the statements within a procedure may use entities previously declared in the same procedure, any enclosed procedure or the enclosing module. Thus the statements of procedures A and B may access any entity previously declared in A or B respectively or declared at the start of the module M. The statements of C may access any entity previously declared in C, A or the module M. In

particular the statements of A may use C, A and B but the statements of B may not use C.

From the point of view of the general scope rule blocks and procedures need not be distinguished. Obviously since a block has no name and is not formally declared a different example to the above is necessary in order to illustrate the rule

```
MODULE M1
  declaration of procedure A
  declaration of X
  .
  .
  BEGIN
    declaration of Y
    .
    .
    statement S1
    .
    .
  END
  PROC A
    declaration of Z
    BEGIN
      .
      .
      statement S2
      .
      .
    END
  END
*END
```

In this case statement S1 may use Y, A and X but not Z. Statement S2 may use Z, A and X but not Y.

There is one important difference between the treatment of procedures and blocks that is relevant to understanding the exception to the general scope rule stated below. This difference is that on entry to a procedure, a 'frame' is created on the procedure stack that contains all the variables

declared in the procedure, together with those declared in the blocks that it contains (and the blocks that they contain and so on), whereas it follows that on entry to a block there is no such action. In effect it is as if the entities declared in a block were actually declared in the enclosing procedure (or module) head, except that access to them is restricted to the statements of the block. This interpretation should be assumed in understanding the restriction stated below regarding the scope of variables.

When nested procedures occur in a module, the only variables available to the statements of a procedure are those previously declared in either the module heading, the outermost procedure of the nest and the procedure in question.

Further restrictions apply to labels. Normally only the labels declared within a block or procedure may be used in the statements of the procedure. A special mechanism exists, as described in Section 9.6, that provides an escape from this restriction.

9.2.8 ‘Statements’ as the Compiler sees them

From a language definition point of view the recursive approach followed above seems best. For example, a procedure definition is regarded as a statement, even though it contains an arbitrary sequence of statements some of which may be further procedure definitions. In the current implementation of MUSL, the compiler takes a different view of statements. The compiler has a main loop, which proceeds along the lines

READ A STATEMENT

IDENTIFY IT

CALL THE ASSOCIATED
TRANSLATION PROCEDURE

Obvious problems arise, on small computers such as the PDP11, if STATEMENTS are very large. Thus the compiler is organised to break certain

potentially long statements into a sequence of statements. For example, a procedure statement is treated by the compiler as the group of statements.

```
procedure heading statement
the statements of the procedure
an END statement.
```

Clearly context information must be remembered across such a group of statements, so that when, for example, the END statement occurs the compiler knows of what it is the end. Even so this feature of the compiler influences the style of fault monitoring, described in Chapter 10, and it is responsible for some of the restrictions mentioned, in subsequent Sections, in connection with specific statement descriptions concerning the use of statement separators and newlines.

9.3 BASIC ELEMENTS OF THE LANGUAGE

9.3.1 Symbols

Each statement in MUSL is a sequence of symbols. These symbols may be names, numbers or delimiters. To compensate for the shortage of suitable characters some delimiters are represented by reserved words. Alternatively an abbreviated form of each reserved word may be used comprising the first two characters of the full name preceded by the \$ sign. The space character has no meaning at the statement level, but it terminates a symbol. Therefore it should be used to separate symbols where ambiguity would otherwise arise. Newline and tab are equivalent to spaces, except where specific restrictions are placed on the use of newlines in certain 'long statements'. The list below gives the full form of each reserved word delimiter.

```
IF, FI, THEN, ELSE, DO, OD, WHILE, WITHIN,
FOR, EXIT, SWITCH, ALTERNATIVE, FROM,
INTEGER, LOGICAL, REAL, SELECT, PROC,
PSPEC, LSPEC, LITERAL, DATAVEC, END,
BEGIN, MODULE, ADDR, SPACE, VSTORE,
LABEL, AND, OR, OF, IS, TYPE, IMPORT.
```

The delimiters INTEGER, LOGICAL, REAL are special in that they can be followed by an integer, giving their size in bits. Thus in effect INTEGER16(\$IN16) LOGICAL8(\$L08) and REAL64(\$RE64), for example,

are also delimiters.

9.3.2 Statement Separators

A ‘;’ is normally used to separate statements, it may be omitted if the statement ends with a reserved word or the following statement begins with a reserved word which cannot occur within a statement. The delimiters that start statements and can also be used within statements are

INTEGER REAL LOGICAL ADDR

Redundant ‘;’s between statements will be ignored, therefore if in doubt, use ; consistently as a terminator.

9.3.3 Comments

The comment symbol is ‘::’. All following characters will be ignored up to the next newline symbol. A comment is an alternative way of terminating a statement and is equivalent to ‘;’.

9.3.4 Names

Names (henceforth referred to as <NAME>) are used to represent variables, types, literals, procedures etc. They must begin with a letter which is then followed by an arbitrary sequence of letters and digits possibly connected by fullstops (period). The latter are allowed as a readability aid and are ignored when names are matched. For example, A X X1 NAME IN.NAME1 and INNAME1 are all examples of valid names, the last two referring to the same item.

9.3.5 Integer Constants

These (henceforth referred to as <CONST>) can be written in various styles. They are all taken to be of TYPE INTEGER (see Section [9.7.2](#)).

```
<CONST> ::= <DEC.INTEGER>!  
           <HEX.CONST>!  
           <CHAR.CONST>!  
           <MULTI.CHAR.CONST>!  
           <NAME>
```


9.3.5.1 Decimal Integer Constants

A decimal integer constant may be optionally preceded by a sign.

```
<DEC.INTEGER> ::= [+!-!<NIL>]<DECIMALS>
<DECIMALS>    ::= <DECIMAL.DIGIT> [<DECIMALS>!<NIL>]
```

For example, all the following are allowed

34, -90, +34, 65535

All the above constants would be appropriate in an INTEGER16 context. They could also be used in an INTEGER32 context, in which case they would be sign extended to 32 bits.

9.3.5.2 Hexadecimal Constants

Hexadecimal constants are used when the bit pattern representing a constant is more significant than the value it represents when interpreted as a number.

They are represented by the ‘%’ symbol followed by a sequence of hexadecimal data, and the notation allows multiple occurrences of the same digit to be specified by a repetition factor expressed as an unsigned decimal integer enclosed in brackets.

```
<HEX.CONST>    ::= %<HEX.SEQUENCE:>
<HEX.SEQUENCE> ::= <HEX.DIGITS> [<HEX.SEQUENCE>!<NIL>]
<HEX.DIGITS>   ::= <HEX.DIGIT> [(DECIMALS)!<NIL>]
<HEX.DIGIT>    ::= <DECIMAL.DIGIT>!A!B!C!D!E!F
```

Hexadecimal constants are right-justified, for example, the binary patterns

1001100110010000 and 11111110

might be expressed as %9(3)0 and %FE respectively. If a hexadecimal constant occurs in context requiring more than the specified number of bits, it is automatically extended by zeros at the ‘left hand end’.

9.3.5.3 Character Constants

These are used to represent the ISO-code of the visible characters. The notation is

`<CHAR.CONST> ::= '<CHARACTER>`

where CHARACTER represents, any printing character except \$, or the character pairs

\$L
\$P
\$N
\$"
\$\$

representing newline, newpage, null, quotes(") or dollar (\$). For example 'A, '1, '\$\$ are equivalent to %41, %31, %24.

9.3.5.4 Multi-character Constants

These are used when it is required to have the ISO-codes for several characters concatenated into one constant. The notation is

`<MULTI.CHAR.CONST> ::= "<CHARACTERS>"`
`<CHARACTERS> ::= <CHARACTER> [<CHARACTERS>!<NIL>]`

Examples are "NAME1", "\$\$RE" which are equivalent to the hexadecimal constants %4E414D4531 and %24245245. Like hexadecimal constants they are right-justified and zero extended on the left when used in a context bigger than themselves.

9.3.5.5 Name Constant

Any name can be defined as a literal and given a constant value (See Subsection 9.5.4). Its use elsewhere is equivalent to using the value explicitly. In fact names can also be assigned to the Character Strings and Real Constants described below.

9.3.6 Character Strings

When general character strings are to be passed as parameters, they should be stored in a vector of bytes and a ‘reference’ or pointer to the vector should be used as the parameter. Procedures that have parameters of this kind will have them declared as ADDR [LOGICAL8] (See Subsection 9.5.2). Since this situation occurs frequently in programs that require to print messages, a special notation is provided that allows the actual character string to be written at the point where a reference to it is required. This notation is

```
<CHAR.STRING> ::= %"<CHARACTERS>"
```

An example of its use would be

```
CAPTION ("%THIS IS AN ERROR MESSAGE");
```

9.3.7 Real Constant

Real constants are used to represent floating point numbers. The number of bytes occupied by the floating point number is determined by its TYPE. Floating point numbers are internally represented as 32 or 64 bit entities.

When real constants are input, the digits up to the decimal point are evaluated as an integer constant and then converted to real. The precision of the conversion is the precision allowed for the (default) integer constant (32 bit integer). The remainder of the constant is evaluated as real. If greater precision is required, the exponent form (+ 1.23456789@8) must be used.

```
<REAL.CONST> ::= [<NIL>!+!-] [<DECIMALS>!<NIL>]  
                [<DECIMALS>!<NIL>] [(@<DEC.INTEGER>!<NIL>)]
```

For example

```
+23 17.3 -90.23 -3.@9 4.3@-4 -.6666 +.5@+3
```

9.4 DIRECTIVE STATEMENTS

All DIRECTIVES start with “*” which is usually followed by the name of the directive, some parameters if appropriate, and they are terminated by ;.

They are used mainly to control the mapping of a module or program and its variables into areas, to control the form of compile time output and to manipulate the environment of the compilation by making calls on MUTL procedures or other library procedures.

```

<DIRECTIVE.STATEMENT> ::= *CODE<CONST>!
                           *GLOBAL<CONST>!
                           *CMAP<CONST>!
                           *STOPC!
                           *INFORM<CONST>!
                           *INIT<CONST>!
                           *TLSEG<CONST><CONST><CONST><CONST><CONST>!
                           *TLLOAD<CONST><CONST>!
                           *TLMODE<CONST> <CONST>!
                           *VTYPE<TYPE>!
                           **<ANY MUSS COMMAND>

```

9.4.1 Areas

Areas are the major logical subdivisions of the store in which a module is to be placed. A module can reference up to 32 areas, which at compile time are known only by number (0-31). They are assigned to actual store locations by use of the MUTL procedures TL.SEG and TL.LOAD. Area 0 is special in that it is the procedure stack, whose placement in store is directly under the control of MUTL. Code is always compiled into the area last specified in the directive

```
*CODE <CONST>
```

If the same area is specified more than once, the successive sequences of code will occur consecutively in the given area.

A precisely similar effect can be achieved with the mapping of global variables (i.e. those declared at the outer level of a module) into segments by use of the directive

```
*GLOBAL <CONST>
```

Area 0 may be specified, in which case, global variables subsequently declared will be on the stack immediately before the first stack frame. Variables declared inside procedures are dynamically allocated on the stack.

9.4.2 Compiler Output

Another pair of directives which are concerned with the output that is produced during compilation, are

```
*CMAP <CONST>
*STOPC
```

The first of the pair causes compile time output to appear on the stream specified by the given constant, and the last stops this output. The CMAP is a compile map giving line numbers and positions in store as various landmarks are passed.

To assist with understanding the action of the compiler, for example, if modifications are required, there is a further directive

```
*INFORM <CONST>
```

This will cause printing of various compiler lists and information depending on the value of the bit significant parameter (CONST) as follows.

<u>Bit</u>	<u>Function</u>
0	Print the encoded itemised statement in hexadecimal, for every statement read
1	Print the text character by character as they are read
4	Print the analysis record of <COMPUTATIONS>
5	Inhibit compile map
Bits 6 – 15	control MUTL output (i.e. they are passed as bits 0 – 7 of the parameter to TLPRINT)

9.4.3 Initialisation

If the MUSL compiler is not running in a full MUSS environment, in which the MUTL system can be initialised and stored by direct calls on MUTL procedures from the command stream of a compile run, facilities for this purpose are necessary in the compiler. In particular it is necessary to call TL and TL.END at the start and finish of the compile run, possibly make several calls on TL.SEG to establish the segments for the compilation in which case it will also be necessary to make calls on TL.LOAD and possibly to call TL.MODE for the compilation of individual modules. Thus the following

directives are provided

```
*INIT<CONST>
*TLSEG<CONST><CONST><CONST>
*TLLOAD<CONST><CONST>
*TLMODE<CONST><CONST>
```

The parameter of *INIT is a bit significant CONST whose bits have the following meaning.

```
bit 0 = 0   the compilation is for a 32-bit machine
bit 0 = 1   the compilation is for a 64-bit machine
bit 1 = 1   the module following is the first of a compile run
bit 2 = 1   the module following is the last of a compile run.
```

If bit 1 = 1 a call will be made on TL and bits 8–15 of CONST will be used as the parameter of TL. If bit 2 = 1 there is no immediate action but the next END encountered will finally result in a call on TLEND.

*TLSEG *TLLOAD and *TLMODE should be followed by the appropriate parameters separated by spaces.

9.4.4 Binary Patching

There is also a *<CONST> directive which allows binary machine instructions to be introduced into critical machine dependent sequences. This is used in MUSS, for example, to implement register dumping and reloading when a process change takes place.

9.4.5 MUSS Library Calls

If the MUSL compiler is running in a MUSS environment, a call of any MUSS library procedure can be forced from the compiler by using in the MUSL source a MUSS command preceded by **. This facility is most often used to manipulate the input/output streams. For example, two separate files can be made to appear as one to the compiler by using the 'IN' command in one file (i.e. '**IN name of file') at the point where the other file is to be included.

Another use of the facility is to make direct calls on the MUTL procedures that control the environment of the compilation.

The command following the `**` up to the next newline is processed by MUSS, not by the compiler, and it should not therefore be terminated by `';` or a comment.

9.4.6 Vstore Type

The VSTORE declarations of Subsection 9.5.10 introduce the idea of a default VSTORE type. This is normally the same as the default integer type but it may be reset by the directive

```
*VTYPE INTEGER32;
```

for example.

9.5 DECLARATIVE STATEMENTS

Every name used in a MUSL statement must be declared before it is used except for the reserved names of the language and label names. The declarations of the language are

```
<DECLARATIVE.STATEMENT> ::= <LABEL.DEC>!
                             <VAR.DEC>!
                             <PROC.DEC>!
                             <LIT.DEC>!
                             <DATA.VEC>!
                             <TYPE.DEC>!
                             <FIELD.DEC>!
                             <SPACE.DEC>!
                             <V.STORE.DEC>!
                             <IMPORT.DEC>
```

In effect a declaration specifies one or more names, defines the kind of entities to which they relate, and where relevant gives their properties. Thus a name declared in a LABEL.DEC is regarded as a label name and a name declared in a PROC.DEC is regarded as a procedure name. The former will have no additional properties but the latter might have properties concerned with the kind and type of its parameters and result.

9.5.1 Label Declarations

$\langle \text{LABEL.DEC} \rangle ::= \langle \text{NAME} \rangle :$

A label declaration provides a way of providing transfer of control to the statement which follows it. The label appears in a ‘– >’(goto) statement. Normally labels are automatically generated by Floccoder and the user will not be directly concerned with them. In effect the label name is a literal whose value is the address of the statement that follows it. Unlike all other declarations the scope of the name declared is the *whole* of the procedure or block in which it is declared. That is, forward references within a procedure or block are permitted, whereas in all other declarations, they are only effective forwards from the place where they appear. Labels may not be referenced in a ‘– >’ statement of an enclosed procedure, since a goto of this kind implies unwinding the procedure stack. However this restriction does not apply to label variables and parameters of type LABEL.

9.5.2 Variable Declarations

$\langle \text{VAR.DEC} \rangle ::= \langle \text{TYPE} \rangle \langle \text{NAME.LIST} \rangle$
 $\langle \text{NAME.LIST} \rangle ::= \langle \text{NAME} \rangle [, \langle \text{NAME.LIST} \rangle ! \langle \text{NIL} \rangle]$

A variable declaration specifies an arbitrary list of names of variables of a given type. They may be scalar or vector variables depending on whether or not the dimension is given in

$\langle \text{TYPE} \rangle ::= \langle \text{SCALARTYPE} \rangle [\langle [\rangle \langle \text{CONST} \rangle \langle] \rangle ! \langle \text{NIL} \rangle]$

Scalar types may be numeric types, type LABEL, the names of user defined types (see later), pointers to scalar or vector instances or either of these, or pointers to procedures. In this latter case the name must be one for which at least a PSPEC and possibly a procedure definition has been previously given.

$\langle \text{SCALARTYPE} \rangle ::= \langle \text{NUMERICTYPE} \rangle ! \text{LABEL} ! \langle \text{NAME} \rangle !$
 $\text{ADDR} [\langle \text{NUMERICTYPE} \rangle ! \langle \text{NAME} \rangle] !$
 $\text{ADDR} \langle [\rangle [\langle \text{NUMERICTYPE} \rangle ! \langle \text{NAME} \rangle] \langle] \rangle !$
 $\text{ADDR} \langle [\rangle \text{ADDR} \langle] \rangle ! \text{ADDR} \text{ ADDR} ! \text{ADDR}$
 $\langle \text{NUMERICTYPE} \rangle ::= [\text{INTEGER} ! \text{REAL} ! \text{LOGICAL}] \langle \text{SIZE} \rangle$
 $\langle \text{SIZE} \rangle ::= 1 ! 8 ! 16 ! 24 ! 32 ! 64 ! 128 ! \langle \text{NIL} \rangle$

For example

INTEGER32	denotes a 32-bit signed integer
LOGICAL8	denotes an 8-bit unsigned integer
INTEGER32[10]	denotes a vector of 10 32-bit integers
ADDR INTEGER32[10]	denotes a vector of 10 addresses of 32-bit integers
ADDR [INTEGER32]	denotes the address of any vector of 32-bit integers
ADDR P1	can be used to indicate the address of a particular type of procedure where P1 is a name defined in a PSPEC

The main numeric types are integer and real, but each may exist in a range of sizes. For type INTEGER the size may be 8, 16, 24 or 32, short operands will be sign extended up to the size of the COMPUTATION in which they appear. If the size indication is omitted the natural size (16/32 bits) of the target machine will be chosen. Type REAL may only be 32, 64 or 128.

The third numeric type (LOGICAL) is similar to INTEGER, but the sign propagation of short operands is suppressed and sizes 1 and 64 are also permitted. It should be noted that LOGICAL is merely a name for unsigned integers. Entities of this type can be used in any INTEGER context. There is no separate LOGICAL mode of arithmetic but the logical operators representing ‘and’, ‘or’ and ‘non-equivalence’ can be used in INTEGER computations.

Variables of type LABEL may have labels (or other label variables) assigned to them. They may then be used as the operand of a ‘goto’ statement.

In the declaration of vectors the CONST should have an integer value > 0 . It specifies the number of elements in the vector and subsequent accesses should regard them as being numbered 0 to CONST-1

When a type is preceded by ADDR, it denotes a pointer to an object of that type. ADDR may also be used as a numeric type to indicate an integer whose size is the address length of the target machine. Similarly a pointer to such an integer or vector of integers is denoted by ADDR ADDR, ADDR[ADDR]. This size may be specified in the mode parameter when the

compiler is called.

Variables which are declared inside a procedure are local to that procedure and are allocated space on its run time stack frame. The variables of a procedure declared at the outer level of a module can be accessed non-locally from any nested procedure. Variables declared at the outer level of a module are *global* to all the procedures of the module. More general non-local access is not permitted. The positions occupied by global variables depend upon the use of the *GLOBAL directive (see Section 9.4).

9.5.3 Procedure Declarations

```
<PROC.DEC> ::= [PSPEC!LSPEC]<NAME>[<PSPEC>!=<NAME>]
<PSPEC>    ::= ([<T.LIST>!<NIL>])[/<SCALARTYPE>!<NIL>]
```

A declaration should normally be given for every procedure, before it is used or defined. This declaration defines the NAME to be a procedure name, indicates whether the procedure is to be given (PSPEC) or found in a library (LSPEC), gives the type of each parameter, and the type of the result (after the /), if it is a function procedure. Only scalar types are permitted as parameters, and results are further restricted to non user defined types and addresses of user defined types.

```
<T.LIST> = <SCALARTYPE>(<T.LIST>!<NIL>]
```

Thus vectors cannot be passed as parameters but their addresses can. Also addresses of user defined types can be passed thereby enabling procedures to place results in the associated variables.

If a group of procedures have identical specifications a full specification must be given for the first then the '='s option can be used in all further PSPECs.

Sometimes it might be appropriate to give a specification for a name of a class of procedures (e.g. TRIGFN) which is not itself a procedure, and to use it to declare the procedures of that class, for example

```
PSPEC SIN = TRIGFN
```

It may also be used to declare pointers to procedures of that particular class,

for example

ADDR TRIGFN

If a user prefers to forfeit some degree of checking, the LSPECs may be omitted and the compiler will automatically generate LSPECs for any undefined procedure name which corresponds to a library procedure name in a currently open library. This option is controlled by a mode bit (see Section 2.5).

9.5.4 Literal Declarations

These declarations allow names to be assigned to various kinds of constant, e.g. INTEGER constants, REAL constants, AGGREGATE constants and POINTER constants. The syntax for each kind of constant is different, thus

```
<LIT.DEC> ::= LITERAL[<INTEGER.LITS>!
                        <REAL.LITS>!
                        <AGGREGATE.LITS>!
                        <POINTER.LIT>]
```

An INTEGER literal may be of any INTEGER type and several may be defined in the same statement. It is the only kind of literal where compile time evaluation of an expression is allowed

```
<INTEGER.LITS>    ::= [ / <INTEGER.TYPE> ! <NIL> ] <INTEGER.LIST>
<INTEGER.TYPE>    ::= INTEGER, INTEGER8, INTEGER16, INTEGER32
<INTEGER.LIST>    ::= <NAME> = <VALUE> [, <INTEGER.LIST> ! <NIL> ]
<VALUE>           ::= <CONST> [ <CONST.OP> <VALUE> ! <NIL> ]
<CONSTOP>         ::= +, -, &, !, *, /
```

Some examples of the declaration of INTEGER literals are

```
LITERAL      L1 = 10, L2 = 20, L3 = L1 - 1;
LITERAL/INTEGER8 A="A", Z="Z", ONE="1";
```

When the TYPE is not explicitly given the default INTEGER type will be assumed.

In practice INTEGER LITERAL declarations are sometimes very long with many names declared in the same statement. Thus the compiler has

been organised to take them a line at a time. As a consequence of this each intermediate line must end with ‘,’. For example

```
LITERAL A = 1,
B = 2;
```

is acceptable but

```
LITERAL A = 1, B
= 2;
```

is not acceptable.

REAL literals are names assigned to floating point values. Thus

```
<REAL.LITS> ::= <REAL.TYPE><REAL.LIST>
<REAL.TYPE> ::= <REAL, REAL32, REAL64
<REAL.LIST> ::= <NAME>=<REAL.CONST>[, <REAL.LIST>!<NIL>]
```

For example

```
LITERAL/REAL      TEN = 10., ONE.TENTH = 0.1;
```

AGGREGATE literals are names assigned to particular instances of user defined types which are composed of a sequence of basic types. The notation for expressing such a value is to give each of the individual basic components separated by ‘\’. Thus

```
<AGGREGATE.LITS> ::= /<TYPE.NAME><AGG.LIST>
<AGG.LIST>      ::= <NAME>=<AGG.VALUE>[, <AGG.LIST>!<NIL>]
<AGG.VALUE>    ::= <CONST>!<REAL.CONST>[\<AGG.VALUE>!<NIL>]
```

For example, if UT1 is a user defined type comprising two integers and a real, then a name ‘X’ might be assigned to a particular instance of UT1 thus

```
LITERAL/UT1      X = 1\ 2\ .135;
```

The final kind of literal definition allows ‘nil’ pointers to be created for any pointer type. The syntax is

`<POINTER.LIT> ::= /ADDR[<ANYTYPE>]<NAME>=;`

For example

`LITERAL/ADDR[LOGICAL8] NIL.STR=;`

defines NIL.STR to have the value of a pointer to an empty vector of bytes.

9.5.5 Data Vectors

A Data Vector is a one dimensional array of literals having a name which can be used like any other vector name in order to access the elements of the vector as operands. However, if the data vector is stored in a read only segment it cannot be altered by the program. If it appears at the outer level of a MODULE it will be placed with the global variables of the module and if it is contained within a procedure it will be stored with the code. Thus the placement of data vectors can be determined by the programmer by preceding them with *GLOBAL directives in the first case and *CODE directives in the second case.

```
<DATAVEC> ::= DATAVEC<NAME>(<SCALARTYPE>)
              <LITERALS>
              END
```

Any number of LITERALS are allowed but they must all be of the same given TYPE. The only permitted pointer type of data vector element is the address of a procedure or data vector (ADDR<NAME>).

```
<LITERALS> ::= <LITERAL>[<[><CONST><]>]<NIL>]
              [<LITERALS>]<NIL>]
```

Obviously the individual literals must be separated and it is usual to use space or newline for this purpose. If a LITERAL is followed by a bracketed <CONST> it will be repeated the CONST number of lines. Only CONSTs with integer values > 0 should be used in this context. For example:

```
DATAVEC NAME($L08)
'J 'A 'C 'K
END:
```

is a VEC called NAME of 4 bytes, and

```
LITERAL COUNT = 4
DATAVEC CONSTS($IN)
1 2 3
10[5]
20[COUNT]
6
END;
```

is a VEC of 13 integers.

Because of the high incidence of character data vectors in some parts of MUSS, for example, as error messages, a special construct is provided to abbreviate lists of character literals. It is "<CHSTRING>". Thus the first example above could be written

```
DATAVEC NAME($L08)
"JACK"
END;
```

9.5.6 Type Declarations

```
<TYPE.DEC> ::= TYPE<NAME> [IS <STRUCTURE>!<NIL>]!
              TYPE<NAME> = <CONST>!
              WITHIN<NAME> IS <STRUCTURE>
<STRUCTURE> ::= <FIELDS>[OR<STRUCTURE>!<NIL>]
<FIELDS>    ::= <TYPE><NAME.LIST>[<FIELDS>!<NIL>]
```

Type declarations allow the user to declare a NAME as a type name and to associate it with a STRUCTURE comprising several fields of simpler types. The fields are given names so that they may be subsequently accessed as operands. Consecutive fields of the same type are declared by giving a list of names after the TYPE. If fields are of different types a TYPE word should appear before each field name. A TYPE.DEC which has an alternative is called a UNION. In all declared instances of a UNION, the compiler will allow space for the largest alternative form, but the onus is on the user to know which alternative is present at any point in time.

It is sometimes necessary to use a TYPE name before giving its declaration.

This may occur, for example, in importing procedures which use exported types. Thus the NIL alternative of a type declaration allows a name to be introduced as a type name. Its full definition must be given later in the usual way.

The last two alternatives of a type definition allow a type to be defined incrementally with several modules adding some fields. First of all a type name, and size in bytes, must be given using the construct

```
TYPE PRB.TYPE = 1000;
```

which states that the type PRB.TYPE will have a size not greater than 1000 bytes. The WITHIN statement which has the same syntax as a normal type definition, may then be used in any module into which the type is imported to add some fields. Only those fields declared in a module will be accessible in that module.

9.5.7 Examples of Variable/Type Declarations

The simplest example of a declaration is where no dimension is given, e.g.

```
INTEGER I,J;
```

In this case I and J will be local variables of type INTEGER. If a vector of elements of type INTEGER is required a dimension must be given thus:

```
INTEGER[25]VECI;
```

In this case the 25 INTEGER elements of VECI will be numbered 0,1,...,24.

New types are created by use of the TYPE declaration, e.g:

```
TYPE IPAIR IS $IN PROP1, PROP2:  
TYPE CONDITIONS IS REAL TEMP, PRESS, VOL;
```

These TYPE names can be used to declare scalar or vector instances of the TYPEs in question, e.g.:

```
IPAIR I1,I2,I3;
```

declares three integer pairs and

```
CONDITIONS [20]U,V;
```

declares two vectors each having 20 elements containing a TEMP, PRESS and VOL component.

A TYPE may have several different fields, e.g.:

```
TYPE IOAREA IS $IN SPN, PID $L08[100]MBUFF
```

declares a TYPE IOAREA with three fields which are the two integers SPN, PID and a vector (MBUFF) of 100 bytes. This might be used to declare an IOCONT vector containing 16 IOAREA's thus:-

```
IOAREA[16] IOCONT;
```

It should be noted that every field has a name which must be unique within the type the purpose of which is to provide a means of accessing it as described in Section 9.5.

9.5.8 Field Declarations

These are duplicate declarations for the fields of structures which allow them to be 'selected' and subsequently accessed by their field name only. The syntax of the statement is given below, but its meaning and use is discussed in Section 9.6.

```
<FIELD.DEC> ::= SELECT<CONTEXT>
```

where CONTEXT is defined in Section 9.6.

9.5.9 Space Declarations

It is convenient to be able to reserve some unmapped store to be dynamically mapped by means of the MAKE function. The SPACE.DEC provides for this.

```
<SPACE.DEC> ::= SPACE<NAME>[<[<CONST><]>!<NIL>]
```

If these declarations appear inside a procedure the space will be reserved on the run time stack otherwise it will be reserved in the current global area (i.e. that last selected by a *GLOBAL directive).

The NAME may subsequently be used as the parameter of a call for the built-in function MAKE. The CONST specifies the required amount of space in bytes and must be an integer > 0. Thus a space of 1 Kbytes called HEAP1 would be declared

```
SPACE HEAP1[1024]
```

The use of the MAKE function and the provision made for ‘garbage collection’ in such spaces is discussed in Subsection 9.6.5.

In some situations in MUSS areas of store arise dynamically. For example, when a file is opened, or a message is received. These can be treated as ‘SPACE’s if a space name declared without parameters is subsequently associated with an area by means of the built in function POSN also described in Subsection 9.6.5.

9.5.10 V-store Declarations

The MUSS ‘ideal machine’ is defined as a set of ‘ideal V-lines’ concerned with the control of peripherals, store management hardware and CPU status. V.STORE.DEC is concerned with their mapping into an actual machine. There are two forms of V.STORE.DEC as follows

```
<V.STORE.DEC> ::= VSTORE <VDEFS>
<VDEFS> ::= <VDEF> [, <VDEFS> ! <NIL>]
<VDEF> ::= <NAME> [<[><CONST><]> ! <NIL>]
               [%<HEXCONST> ! <NAME>]
               [<<><NAME> ! <NIL>]
               [<>><NAME> ! <NIL>]
```

One option in V.STORE.DEC provides a means for associating an absolute address with an ideal V-line. It uses the CONST option where the CONST must have an integer value giving the address of the V line in bytes and its type will be taken to be the default VTYPE (see Subsection 9.4.6). For example

```
VSTORE LPTSTATUS %3FF4C;
```

associates LPTSTATUS with the byte address %3FF4C. In cases where the machine VSTORE does not correspond to the ideal machine VSTORE, it is necessary to map the ideal VSTORE onto a variable and to emulate the special actions of the ideal VSTORE by using PREPROCs and POSTPROCs.

Thus another form of VDEC is

```
VSTORE V1 PSEUDOV1 <PROC1>PROC2
```

Here the VSTORE variable V1 is mapped onto the variable PSEUDOV1 (and takes its type), and the names which (optionally) follow are the names of procedures. If the first procedure name is present (e.g. PROC1 above) it will be called before a read access on V1 and the second (PROC2) will be called after a write access to V1. The TYPE of VSTORE variables may be machine dependent but it will usually be type INTEGER, although if a VSTORE variable is mapped into another variable it takes the TYPE of that variable.

Some VSTORE entities, for example, page address registers occur naturally as vectors. In these cases a dimension should be given as for vectors of ordinary variables.

A procedure used as a PREPROC or POSTPROC is a restricted form of procedure. It may not have local variables, parameters or a result. However, the code it contains may use the automatically declared name VSUB, providing it has not been redeclared, to obtain the subscript in the case where it relates to a vector of V-lines. In effect the VSTORE declaration serves as the 'PSPEC' of the 'PROC's. Hence it should precede the PROCs and be at the same block level. In particular MUTL implementations the restrictions may be more severe (e.g. language constructs that cause the compiler to generate local variables must be avoided).

9.5.11 Import Declarations

```
<IMPORTS> ::= <IMPORT><IMPORTS>!<NIL>
<IMPORT> ::= <PROC.DEC>!<VAR.DEC>!<IMPORT.DEC>!<TYPE.DEC>
```

In general a module may import any entity which is exported from another module. When the imported entity is a procedure or variable a full duplicate specification must be given before the module heading.

A module sometimes uses imported entities without requiring a full specification to precede the module heading. This applies to TYPEs, VSTORE, LABELS and LITERALs. Thus the following special IMPORT statement is provided

```

<IMPORT.DEC> ::= IMPORT<KIND><IMP.LIST>
<KIND>      ::= [LITERAL!VSTORE] [<NUMERICTYPE>!<NIL>]!
               TYPE!LABEL
<IMP.LIST>   ::= <NAME> [<[><]>!<NIL>] [, <IMP.LIST>!<NIL>]

```

A type may be specified for an imported LITERAL or VSTORE and the NAME[] construct is used to indicate when a vector instance of VSTORE is to be imported.

If a type name is imported in the above manner only the type name and not its fields are accessible in the module into which it is imported. A full duplicate type definition must be given before the module heading if use is to be made of the field names of the type. Similarly if procedures or variables are imported, full duplicate definitions are necessary. Clearly any entity that is imported into a module must be exported from another module, and modules related in this way must be loaded in the same load run, or compiled in the same compile run if a generate executable binary option is used.

9.6 OPERAND FORMS

```

<OPERAND>    ::= [↑!<NIL>] <VARIABLE> [OF<CONTEXT>!<NIL>]!
               <CONST>!
               <NAME> [↑!<NIL>] ([<P.LIST>!<NIL>])!
               (<COMPUTATION>)!
               (<COND.COMP>)!
               <BUILT.IN.FUNCTION>
<VARIABLE>   ::= <NAME><SUBSCRIPT> [↑<SUBSCRIPT>!<NIL>]
<CONTEXT>    ::= <VARIABLE> [OF<CONTEXT>!<NIL>]
<SUBSCRIPT>  ::= <[><COMPUTATION><]>!<NIL>
<P.LIST>     ::= <COMPUTATION> [, <P.LIST>!<NIL>]
<COND.COMP>  ::= IF<CONDITION> THEN<COMPUTATION>
               ELSE<COMPUTATION>

```

<COMPUTATION> and <CONDITION> are defined in Subsection 9.7.1 and Subsection 9.7.3.

The operand capability of MUSL is moderately complicated and warrants some discussion. Inevitably the examples given reflect the structure of the COMPUTATIONs described in the next Section but their meaning should

be self evident.

9.6.1 Variables

Considering first the case where no CONTEXT is given, VARIABLES which are scalars are represented by names, which are declared as described in Section 9.4. They may be Local, Non-local or Global. Those which are vector elements are represented by the vector name followed by a subscript. The subscript (or index) is the result of a computation which gives the required element number. For example, A[0] denotes the first element of a vector A and A[9] denotes its 10th element. It follows that A[I-1] denotes the Ith element. If a variable contains the address of the required variable it must be followed by "↑" to explicitly dereference it. Any variable so dereferenced might have been a pointer to a vector in which case a further subscript is required. For example if the NAME in the definition of VARIABLE refers to the address of a vector, and for example, it has been declared

$$\text{ADDR}[\text{INTEGER}] \text{ name1};$$

and the address of a suitable vector has been assigned to name1, then an element in the vector is accessed thus:

$$\text{name1}\uparrow [\text{subscript}]$$

Whereas if it were the NAME of a vector of addresses of integers, declared as

$$\text{ADDR INTEGER}[10] \text{ name2};$$

any one of the integers addressed could be accessed thus:

$$\text{name2} [\text{subscript}]\uparrow$$

If a reference to a variable is to be created then its name is preceded by an "↑". Thus

$$\uparrow\text{name1}\uparrow[4] \Rightarrow \text{name2}[3]$$

computes the address of the 5th element of the vector of integers whose address is given by name1 and assigns it to the 4th element of the vector of addresses name2.

When a variable is a field of an aggregate type its CONTEXT must also be given. E.g

SPN OF IOCONT[I]

If it is required to access an element of a vector which is itself a field in an element of a vector, for example an element of the MBUFF defined in Section 9.5, the notation is:

BUFF[J] OF IOCONT[I]

meaning the Jth+1 element of the MBUFF in the Ith+1 (aggregate) element of IOCONT.

In general aggregate types can be nested to an arbitrary depth, hence, CONTEXT is defined recursively. It should be noted that the last name given in the specification of a variable refers to an instance of a declared object, whereas the preceding names are field names. Any of these names might be subscripted and/or dereferenced.

The repetition of the full specification, when a series of operations are performed on a variable embedded in a STRUCTURE, is tedious. The FIELD.DEC described in Subsection 9.6.3 allows them to be accessed through their field names only.

9.6.2 Constants

These were described in Section 9.3. Any of the usual forms are permitted but they must be type compatible with the context in which they appear (see Subsection 9.7.2).

9.6.3 Field Declarations

The syntax was given in Section 9.4. Its use is illustrated in the example below.

SELECT IOCONT[I]

serves two functions. It causes the address of the element of IOCONT specified by I to be noted and it implicitly declares all the fields of IOCONT to be accessible by their field name only.

Given this, and assuming SPN, PID and MBUFF are fields of IOCONT,

a statement such as

```
IF SPN OF IOCONT[I] = 0 THEN
    FOR J< 100 DO 0 => MBUFF(J) OF IOCONT[I]OD
FI
```

could be rewritten

```
IF SPN = 0 THEN FOR J<100 DO 0 => MBUFF [J]OD
```

The addresses of the selected fields are computed at the time the declaration is processed. Hence, subsequent changes in the values of variables which appear as subscripts of the selected fields will be ineffective.

9.6.4 Procedure Calls

Procedures are called by giving their name (or a dereferenced pointer to a procedure) followed by a P.LIST enclosed in brackets. The parameters must be COMPUTATIONs which yield values of the correct type, although the usual automatic type conversions that apply in COMPUTATIONs may be assumed. For example, an INTEGER can be substituted for a REAL. Type and type conversions are discussed in Subsection [9.7.2](#).

Procedures which return results do so by assigning them to the name of the PROC as described earlier.

9.6.5 Built-In Functions

Several built-in functions have already been mentioned but the significance of "built-in" has not been explained. Their textual representation is that of a procedure. They are built into the compiler because their implementation uses facilities not available at the user level. These fall into three main groups.

- direct manipulation of addresses
- variable TYPE parameters
- interpretive entry to libraries.

A complete list of the built-in functions is

```

MAKE(ANY TYPE, $IN, [ANY SPACE!A BYTE ADDRESS!<NIL>]
                                     )ADDR OF GIVEN TYPE
MKDT(<COMPUTATION>,<COMPUTATION>,[0|1|ANY SPACE])BYTE ADDRESS
PART(ADDR OF A VEC, $IN, $IN)ADDR(OF A VEC OF LIKE TYPE)
SIZE(ADDR OF A VEC)$IN
POSN(ANY SPACE, A BYTE ADDRESS)
BYTE(<COMPUTATION>)$IN(A BYTE ADDRESS)
LPST([ANY BASIC TYPE, COMPUTATION|<NIL>])
LENT(INTEGER COMPUTATION, ANY BASIC TYPE)GIVEN BASIC TYPE
MVEF(<SCALAR TYPE>, A BYTE ADDRESS [, <COMPUTATION> | <NIL>])
                                     GIVEN SCALAR TYPE

```

9.6.5.1 Dynamic Allocation of Store

The built-in function MAKE exists to allocate space at runtime for an object in a space previously created by a SPACE declaration. The function has three parameters. The first is a type name, and the second is a number. Space for the specified number of objects of the given type is reserved. The third parameter of MAKE specifies the name of the SPACE that is to be used.

The function yields a value of type ADDR of type of object created. If the number of objects required is the constant 0 a pointer to a scalar is yielded, otherwise a pointer to a vector instance of the objects is yielded.

Thus:-

```
MAKE (REAL,0,HEAP1)
```

generates space for a REAL value in the space specified as HEAP1 and yields its address.

This implies that associated with HEAP1 is an index that keeps track of the amount of space actually in use. Obviously the need might arise to note and reset this index and this is achieved by using the name of the space as an integer variable.

In fact the third parameter of MAKE can be omitted and in this case the run time stack frame of the procedure will be used, or it can be an explicit byte address.

The function MKDT is used to allocate space for user defined types when

information about the type is only known dynamically. This function is somewhat specialised and may be ignored by normal users. It is typically used to allocate space on the parameter stack when implementing interpretive procedure calling; users should have a thorough understanding of the type and procedure calling mechanism of the MUTL target language of Chapter 18 as use of the storage allocated requires an understanding of its physical layout.

MKDT has three parameters, the first is an address length integer giving the alignment requirements of the object being created, (this may differ depending on the area in which the storage is allocated – e.g. on some machines all parameters are aligned on a 4 byte boundary). The second parameter, also an address length integer, gives the size of the required object in bytes. In general, this information about the alignment and size of objects must be generated by the MUTL system or be extracted from library information about types (see Chapter 7). The final parameter specifies the area in which the storage is to be allocated. The explicit literals 0 and 1 have a special meaning in this context. 0 means use the run time stack frame and 1 means allocate storage on the parameter stack. If neither of these last two cases holds the parameter must be the name of the SPACE that is to be used.

Thus:-

MKDT(I, J, 1);

would allocate space on the parameter stack for an object aligned on a byte boundary given by I and of size J bytes.

9.6.5.2 Address Manipulation

Clearly operands have addresses but normally the programmer is only concerned with their names or with the names of ADDRESS type variables. Addresses may be generated or dereferenced by means of the "↑" operator described earlier. However, certain exceptional requirements arise in system programming which make it necessary to manipulate addresses more directly. A simple example occurs when a character string has to be read, stored and returned as a result. In this case a global or parametric byte vector of adequate length could be used to store the characters and the built-in-function:

PART(<COMPUTATION>, <COMPUTATION>, <COMPUTATION>)

is available to generate the appropriate result to return. The first COMPUTATION should yield the address of a vector and the next two COMPUTATIONS give the first and last subscripts of the partition whose address is required. As the parameters may be evaluated in an implementation dependent order they should have no side effects which would affect the values of the other parameters.

Again in the context of parametric vectors it might be necessary to discover the number of elements that it contains. The function

SIZE (<COMPUTATION>)

yields the size of the vector whose address is yielded by the COMPUTATION.

Another requirement arises as a result of the Operating System allowing segments to pass from one virtual machine to another, either as files or messages. This means that a program can acquire information at a particular address unknown to the compiler. In this case usable addresses for the objects can be obtained from the built-in function MAKE providing the area of store can be treated as a SPACE. The built-in function

POSN (<SPACENAME>, <ADDRESS>)

associates the given SPACE name with the area of store whose byte address is given as the second parameter. The first parameter is the name of the SPACE and the second is its address in byte units. Alternatively, for similar reasons a byte address can be used directly in MAKE. Byte address of variables can be obtained by using the function

BYTE (<COMPUTATION>)

The COMPUTATION must yield a result of pointer type.

9.6.5.3 Interpretive Library Calls

These facilities are required by, for example, a command language interpreter and need not be understood by a normal user. A normal (compiled) procedure call is broken down by a compiler into several steps. These are

notionally

```

prepare to call a procedure
stack the first parameter
stack the second parameter
.
.
stack the last parameter
enter the procedure
assign result

```

Further code might be interleaved with these to compute the parameters. In order to facilitate interpretive procedure calls these same steps are made available individually as built-in functions. They are

```

LPST ()
LPST (TYPE,COMPUTATION) :: For each parameter
LENT (procedure index, TYPE)

```

The procedure index is the integer result returned by the library procedure FINDN. The LENT procedure is used to enter the procedure indicated by the FINDN index given to it. The TYPE (or 0) is the type of the result of the given procedure and if given then this call on LENT may be followed by an assignment to a variable of that type. Some other related library procedures are usually required in order to make an interpretive procedure call. These return information about the number and types of parameters etc., and are described in Chapter 7.

If a parameter in an interpretive procedure call is of a user defined type then space for it is created by using the MKDT function which will yield the byte address of the storage allocated. The command language interpreter can then fill in the fields of basic types which are the components of the user defined type using the MVEF function. The user of this function will need an understanding of the physical layout of fields within types as effected by MUTL (see Chapter 18).

The first parameter of MVEF gives the type of the field under consideration. The second parameter gives the byte address of the field within the storage used for the user defined type, this must be computed from the address of the entire type (yielded by MKDT) and the displacement of the

field within it (this information would be generated by MUTL or the library procedures of Chapter 7). The third parameter gives the value to be stored in the field and must match the type given in the first parameter.

Thus:

```
MVEF ($IN, POSITION, CURR.VAL);
```

will store the \$IN value CURR.VAL in the \$IN field at the byte address given by POSITION.

If the final parameter is omitted the function may also be used to return the value of the field at the address specified in the given type.

9.7 IMPERATIVE STATEMENTS

The Imperative Statements include both:

Computational Statements
and Control Statements

Computational Statements may be conditional, unconditional, repeated WHILE some CONDITION holds or FOR a given number of times (any of which may be zero times), thus:

```
<IMPERATIVE.STATEMENT> ::= <COMP.ST>! <CONTROL.ST>
  <COMP.ST>           ::= <COMPUTATION>!
                        <IF.ST>!
                        <WHILE.ST>!
                        <FOR.ST>
  <CONTROL.ST>        ::= <GO.ST>!
                        <SWITCH.ST>!
                        <ALT.ST>!
                        EXIT
  <IF.ST>              ::= IF<CONDITION><ACTION>
  <WHILE.ST>           ::= WHILE<CONDITION>
                        DO<STATEMENTS>OD
  <FOR.ST>             ::= FOR[<NAME>! <NIL>] <COMPUTATION>
                        DO<STATEMENTS>OD
  <ACTION>             ::= ,<GOST>! THEN<STATEMENTS><ELSECL>FI
  <ELSECL>            ::= ELSE<STATEMENTS>! <NIL>
```

CONDITIONS also appear in conditional control statements and are described in Subsection 9.7.3. In the WHILE statements the CONDITION is repeatedly evaluated. Each time it yields a true result, the imperative statements between the following DO and OD are obeyed. On the first occasion that the condition yields false, control passes to the next statement after the OD. The FOR statement provides for a similar set of imperative statements to be executed a given number of times as specified by the control COMPUTATION which is evaluated once at the start. As the repetition proceeds a counter is maintained which starts from zero. When it reaches the specified value the repetition stops. If this "control variable" is to be used during or after the loop an INTEGER variable name must be given in the NAME. On completion of a FOR loop the value of the 'control variable' will be the value of the control COMPUTATION.

9.7.1 Computations

A computation is expressed as a sequence of operators and operands thus:

$\langle \text{COMPUTATION} \rangle ::= \langle \text{OPR.OPD.SEQ} \rangle$
 $\langle \text{OPR.OPD.SEQ} \rangle ::= \langle \text{OPERAND} \rangle [\langle \text{OPERATOR} \rangle \langle \text{OPR.OPD.SEQ} \rangle$
 $! \langle \text{NIL} \rangle]$

Every COMPUTATION is an expression and the final result which occurs after complete evaluation of the OPERATOR OPERAND sequence (from left to right) is the VALUE of the expression. This value is discarded if the computation is a statement in its own right.

The OPERATORS are the following

+	meaning add
-	meaning subtract
*	meaning multiply
/	meaning divide (rounding towards 0 for integer result)
&	meaning logical "AND"
!	meaning logical "OR"
- =	meaning logical "NON-EQUIVALENCE" (or exclusive OR)
- :	meaning reverse subtract
/ :	meaning reverse divide
+ >	meaning ADD into store
- >	meaning SUBTRACT from store
* >	meaning MULTIPLY into store
/ >	meaning DIVIDE into store
& >	meaning AND into store
! >	meaning OR into store
- ==>	meaning NONEQUIVALENCE into store
=>	meaning assign to
- :>	meaning reverse subtract from store
/ :>	meaning reverse divide into store
<< -	meaning left logical shift
- >>	meaning right logical shift

When an OPERATOR is followed by ">" (excepting of course - >>, << - and ==>), first the OPERATOR is applied (reversed if not commutative) to the current result and the following OPERAND, then this new result is assigned to the following OPERAND and finally the new result is carried forward to the next stage of the COMPUTATION. For example, 1- >X will decrement X and generate a result "X-1". Note that some operators namely &, !, - =, & >, ! >, - ==>, << -, - >> are not available in floating point mode.

General computations are not permitted on variables of aggregate or pointer types. They may, however, be moved and compared, but in the case of aggregates not both in the same statement. For example:

```
AGG1 ==> AGG2 ==> ACG3
$IF AGG1 /= AGG2 /= AGG3 $TH
```

are acceptable.

`$IF AGG1 /= AGG2 => AGG3 $TH`

is not acceptable.

9.7.2 Type

The TYPE of arithmetic used in evaluating a computation is determined by its operands. In any COMPUTATION (or COMPARISON) the TYPE and size of the operands must be compatible. Obviously if they are all the same, they are compatible, but there will be automatic type and size conversion in some other cases. (INTEGERS are always sign extended when converted and REALs are converted to INTEGERS by taking their integer part).

For any computation the compiler determines a computation type by looking ahead up to and including the first ‘assigned to’ operand, or the end of the computation if this occurs first. The operand of the largest type determines the computation type. The possible computation types are INTEGER (size 16/32/64), REAL (size 32/64/128), user defined types and the permissible address types (any REAL type is considered larger than any INTEGER type). However, the operations available are in some case restricted as shown in the table below: (/ indicates allowed, x indicates not allowed).

TYPES	REAL(32/64/128)	INTEGER(16/32)	INTEGER(64)	USER/ADDR
+	/	/	x	x
-	/	/	x	x
*	/	/	x	x
/	/	/	x	x
&	x	/	/	x
!	x	/	/	x
- =	x	/	/	x
- :	/	/	x	x
/ :	/	/	x	x
+ >	/	/	x	x
- >	/	/	x	x
* >	/	/	x	x
/ >	/	/	x	x
& >	x	/	/	x
! >	x	/	/	x
- =>	x	/	/	x
=>	/	/	/	/
- :>	/	/	x	x
/ :	/	/	x	x
<< -	x	/	/	x
- >>	x	/	/	x

Type conversions are permitted only as follows

OPERAND TYPE	COMPUTATION TYPE							
	INTEGER	16	32	64	REAL	32	64	128
LOGICAL1		/	/	/		/	/	/
LOGICAL8		/	/	/		/	/	/
LOGICAL16		/	/	/		/	/	/
LOGICAL24		/	/	/		x	x	x
LOGICAL32		/	/	/		/	/	/
LOGICAL64		/	/	/		/	/	/
INTEGER8		/	/	/		/	/	/
INTEGER16		/	/	/		/	/	/
INTEGER24		/*	/	/		/	/	/
INTEGER32		/*	/	/		/	/	/
REAL32		/*	/*	/*		/	/	/
REAL64		/*	/*	/*		/	/	/

The conversions marked ‘/*’ are permitted but they may generate arithmetic overflow

If an ‘assigned to’ operand is less than the type of the computation, a type conversion will be applied, after the computation has been evaluated. This type converted result is the value carried forward as the first operand when further operators and operands follow. If the expression contains further assignment this process is repeated for each. For example, given

INTEGER32	I,J
LOGICAL64	LL
REAL32	R1
REAL64	RR2

I+J-1 =>R1	will be evaluated in 32-bit real mode
I+J+R1 =>RR2	will be evaluated in 64-bit real mode
R1*RR2 =>J	will be evaluated in 64-bit real mode but converted to 32-bit integer mode
I<<-8!J<<-8=>LL	will be evaluated in 64-bit integer mode

Parenthesised computations yield an operand of the ‘final’ type used inside the parentheses. The result is then converted, if necessary to the type of the expression that contains it. Thus in

$$(I+J-1) => R1$$

the arithmetic would occur in INTEGER32 mode and the result would be converted to REAL32 before being assigned to R1.

In a TEST involving COMPARISONs each computation is evaluated in a type not less than that of the ‘largest’ computation. For example

$$CH() << -8!CH() << -8!CH() << -8!CH() << -8!CH() << -8 = NAME$$

will be evaluated in INTEGER64 mode even if CH delivers a LOGICAL8 result providing NAME is LOGICAL64. For the above purpose the type of each computation is taken to be its final type.

However, in those computations which begin in a type larger than the comparison type, the evaluation will begin in this larger type. For example,

$$LL - >> 32 => I = J$$

is a comparison of type INTEGER32, but LL will be loaded and shifted in 64-bit integer mode before being converted to INTEGER32 type.

Parameter expressions are evaluated in a type of arithmetic consistent with the expression being followed by an assignment to a variable of the type of the formal parameter.

9.7.3 Conditions

<CONDITION>	:= <TEST>[<LOGOP><CONDITION> !<NIL>]
<TEST>	:= <[><CONDITION:><]> !<COMPARISON>
<LOGOP>	:= OR!AND

The simplest form of CONDITION is a TEST in the form of a COMPARISON applied to the results of two or more COMPUTATIONs. The syntax for COMPARISON is:

```

<COMPARISON>      ::= <COMPUTATION><COMPARE.LIST>
<COMPARE.LIST>    ::= <COMPARATOR><COMPUTATION>
                    [<COMPARE.LIST>!<NIL>]
<COMPARATOR>      ::= =!/ =! <<>!<>>!=<<>!<>>=

```

This allows several COMPARISONS to be made against the same RESULT. They must all be satisfied for the CONDITION to be satisfied.

Examples of simple conditions are:

```

X = Y
X-1 /= Y/Z
INSYM() =< Z >='A (meaning the same as below)

```

More complicated conditions would use AND and OR, e.g.

```

INSYM() => SYM =< Z AND SYM >= 'A
X=Y AND Z=10 OR P /= 4

```

The AND symbol has greater binding power than OR hence the following parenthesis is implied:

```
[X=Y AND Z=10] OR P /= 4
```

Conditions are evaluated left to right and the evaluation ceases when the status of the condition is known. This is when a test that succeeds is followed by "OR" or one that fails is followed by "AND".

9.7.4 Control Statements

```
<GO.ST>      ::= - ><NAME>
```

Any basic statement in a program can be labelled thus defining a label name. A GOTO statement may reference these label names or OPERANDS which are of type LABEL. Usually, however, the labels and '- >'s are generated by Flocoder not the programmer. Non-local '- >'s are permitted only where the name is a label variable or parameter of type LABEL normally concerned with error recovery.

The switch statement also uses labels. Its syntax is

<SWITCH ST> ::= SWITCH <COMPUTATION> <N.LIST>;

Here the value of the computation decides which NAME is selected in the N.LIST (value 0 selecting the first) and a GOTO that NAME is obeyed. The NAMEs must be locally defined LABELs.

An alternative statement specifies a COMPUTATION and a list of STATEMENTS thus:

<ALT.ST> ::= ALTERNATIVE <COMPUTATION>FROM
 <BLOCKS>
 END

If <COMPUTATION> yields a value N, the N'th <BLOCK> is executed. Blocks are numbered starting with zero.

9.7.5 Implicit Blocks

Blocks were introduced in Subsection 9.2.6 and the scope restrictions they impose in Subsection 9.2.7. Some groups of STATEMENTS are treated as if they constitute a block even though explicit BEGINS and ENDS are not given. They are the STATEMENTS used in

IF.ST
WHILE.ST
FOR.ST
and ALT.ST

One important effect of this is that GOTOs entering or leaving such statement groups are not permitted.

9.7.6 Examples of a Procedure Definition

As an illustration of a complete sequence of MUSL statements, consider the following procedure for reading a decimal integer.

Its specification is

```
PSPEC IN.I()/INTEGER;
PROC IN.I;
INTEGER SIGN, CH, ANS;

WHILE IN.CH() => CH =< " " DO OD::IGNORE SPACES

IF CH = "-" THEN::DEAL WITH OPTIONAL SIGN
    1 => SIGN;
    IN.CH() => CH;
ELSE
    0 => SIGN;
    IF CH = "+" THEN
        IN.CH() => CH;
    FI
FI
FI

IF CH - "0" => ANS >= 0 =< 9 THEN
    WHILE IN.CH() - "0" => CH >= 0 =< 9 DO
        ANS * 10 + CH => ANS;
    OD

IN.BACKSPACE (1);
IF SIGN = 0 THEN
    ANS => IN.I;
ELSE
    0 - ANS => IN.I;
FI

ELSE
    ENTER.TRAP (6,8);
FI

END
```

MUSS

USER MANUAL

VOLUME 1

CHAPTER 10

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

10 PROGRAM DEVELOPMENT TOOLS

This chapter is concerned with the detection, signalling and analysis of errors, and, more generally, with the analysis of program behaviour. In this sense an error relates to the failure of some piece of hardware or software to carry out the task it has been called upon to do.

There are two possible approaches to the problem of signalling detected errors to a calling program. One is to set an error code into a (global) status variable, and then to return to the calling procedure. Alternatively the calling procedure may be “trapped” – i.e. forced into an error handling procedure which it has previously nominated to deal with the condition. Obviously, each method has its advantages: the former is often easier to use, particularly for errors which are expected to occasionally occur (e.g. a user miss-typing a command parameter); on the other hand, the trapping method is potentially more efficient since it eliminates the need to check the status after each action which might cause an error. MUSS provides for both of these methods, and allows the programmer to select which is to be used for particular classes of error. However, the norm for user programs is for errors to cause entry to traps, and a significant part of this chapter describes a runtime diagnostic mechanism which can be used to intercept the error traps and ascertain their cause.

The trapping and diagnostic mechanism can also be exploited as the means for generating flow analysis data for a program, and this is described later in this chapter.

10.1 CLASSIFICATION OF ERRORS

For convenience of recovery, errors are grouped into a number of error classes, and the recovery action may be specified separately for each error class. Within the classes, individual errors are identified by an error code; a complete list is given in [Appendix A2](#).

The basic system distinguishes sixteen different error classes numbered 0–15. The first ten of these are system defined; classes 10–15 are available for use by applications software. The ten system-defined error classes are:

- 0 Program Faults – this class consists mainly of errors detected by hardware during instruction execution – for example, arithmetic overflow, access to illegal store addresses, etc.
- 1 Limit Violation – these errors are signalled by the system kernel when a process exceeds one of its stated resource limits for example, processing time.
- 2 Timer Runout – these are not strictly errors at all, but are exception conditions handled by the same trapping mechanism. The process may request to be informed after it has used a specified amount of processing time. The mechanism is used by the basic system to obtain an “early warning” of the processing time limit violation, by requesting a timer interrupt shortly before the actual time limit is exceeded.
- 3 External Interrupt – these also are not strictly errors, but are handled by the same mechanism. Two forms of external interrupt exist. The first is an interrupt triggered by some other process, usually a device controller. and its main use is to implement the “break-in” facility for interactive jobs. The second occurs when a message arrives on a message channel where the channel status has been set appropriately (see Chapter 15).
- 4 Organisational Command Error – the organisational command procedures which form the interface with the MUSS kernel all signal errors via this error class.
- 5 Input/Output Initialisation Errors – errors in the definition or organisation of input/ output streams are included in this error class
- 6 Other Input/Output Errors – the errors in this class occur during the use of an input/output stream, rather than its definition. They are usually indicative of faulty data on an input stream rather than a faulty program, for example, receiving non-numeric data in a context where numeric data is expected.

- 7 Job Control Command Errors – this class is reserved for errors detected by the job control command interpreter
- 8 Programming Language Runtime Errors – this class consists of language dependent errors, such as accessing outside the bounds of an array or calling an undefined procedure. To the user they are logically an extension of the class 0 program fault errors.
- 9 Library Utility Errors – all errors detected by the major user facilities of the basic system (i.e. those described in this Manual, such as EDIT, TXT) fall into this class.

As far as the ordinary user is concerned the procedures of the next section are adequate for handling traps. However, error classes 0, 1, 2 and 3 are in fact detected inside the operating system and the following commands relate to their interception.

1) SET.INT.TRAP (INTEGER, ADDR PROC)

This command sets the address of the specified interrupt trap (P1 masked in the range 0 – 3) to be the procedure P2.

2) READ.INT.TRAP (INTEGER)

This command reads the procedure address associated with trap P1 and returns it in PW1.

Normally these ‘interrupt’ traps will be set to a built in procedure which allows them to be treated like all other trap classes. It is only special system procedures such as the profile mechanism of the runtime diagnostic system which need to treat them specially.

10.2 PROCEDURES FOR TRAP HANDLING

Although there are sixteen trap classes, there is only one current trap procedure. In order that the trap procedure may identify the error two parameters are stacked on entry. The first gives the trap class and the second gives the reason code. In fact there is a stack of trap procedures and new entries can be ‘pushed’ on to this stack by the SET.TRAP procedure.

Whenever a trap is entered the top entry is ‘pulled’ from the stack and a procedure call is made into the procedure which it identifies. Entries may also be removed from a stack by the REMOVE.TRAPS procedure. If the stack is empty at the time of an attempted entry to the trap, the ABORT procedure will be called.

Two options are available to a trap procedure. Firstly, it may deal with the reason for the trap and then execute RETURN.FROM.TRAP which will allow the program interrupted by the trap entry to continue. This option might be appropriate, for example, in the case of periodic timer traps. Secondly, the trap procedure may inspect the class and reason parameters and in the case of some reasons prefer to allow the previously stacked trap procedure to take over. This action can be achieved by calling RE.TRAP.

1) SET.TRAP (ADDR PROC) / INTEGER

This procedure nominates P1 as the trap procedure to be added to the entry stack for traps. It should be the address of a procedure with two integer parameters for the error class and code. It returns the number of entries on the trap stack prior to setting the trap.

2) REMOVE.TRAPS (INTEGER) / INTEGER

This procedure removes all entries on the trap stack except the first P1. It has no action if the number of entries on the trap stack are less than this, except in all cases it returns the number of entries -1 remaining.

3) SET.RECOVERY.STATUS (INTEGER, INTEGER)

This procedure is used to control whether the “trapping” or “status” mechanism is to be used for errors of class P1. P2 specifies the mechanism to be used: zero means trapping, 1 means status.

The global status variable used for returning is usually referred to as PW0. On most systems it will occupy the first 32 bits of the stack segment as described in Section [2.3.3](#).

4) READ.RECOVERY.STATUS (INTEGER) / INTEGER

This procedure returns the current recovery status for error class P1.

5) ENTER.TRAP (INTEGER, INTEGER)

This procedure signals the occurrence of the error with class P1, code P2. Depending upon the recovery status in force for error class P1, the error will be signalled either via a trap entry or via the global status parameter PW0.

6) RETURN.FROM.TRAP (INTEGER)

This procedure enables a process which has entered a trap to resume from the point where the trap occurred. It is equivalent to a normal procedure return, except it also causes the register values to be restored to what they were at the time of entry to the trap procedure. Thus trap procedures which, for example, deal with external interrupts can be thought of as ‘interrupt’ procedures which do not interfere with the correct operation of a program. The tracing facilities of the runtime diagnostic mechanism utilise this feature. Thus a parameter P1 is provided which when set to one causes an instruction trace trap (class 0) to occur after one further instruction has been executed at the return address. It should be noted that entry to a trap procedure always removes it from the trap stack. This means that trap procedures which expect to be entered repeatedly must always reset the trap before returning.

7) RE.TRAP

Frequently a user defined trap procedure is only intended to intercept certain error codes. Unwanted error codes are then passed on to the original trap procedure by the user trap procedure calling RE.TRAP. The effect of RE.TRAP is to remove the activation frame of its caller and then to call the procedure specified by the previous trap stack entry (also unstacking it).

8) ABORT (INTEGER, INTEGER)

This is the procedure which will eventually be called if all the trap procedures are themselves trapped. It will terminate the process.

10.2.1 Notes on the Handling of Errors and Traps

- 1) Error classes 0-3 are detected within the operating system kernel, and are always handled via the trapping mechanism (SET.RECOVERY

STATUS should not be used).

- 2) If the “status” form of error recovery is active, then an appropriate error code is set into the global status variable PW0, and the ENTER.TRAP procedure returns immediately. Thus, in cases where status recovery may be in use, calls of ENTER.TRAP should always be followed immediately by a return to the calling program which may then inspect the status variable. Generally, use of the status form of recovery is only recommended in small localised code sequences.
- 3) The status variable PW0 is encoded as two bytes, the most significant giving an error class, and the least significant an error code.
- 4) Organisational command errors (class 4) are initially signalled by a negative error code in PW0. Hence, software calling organisational commands directly should assume the “status” form of error recovery. The command interpreter maps the negative fault reason into a standard form of trap.

10.3 PROCEDURES FOR FAULT MONITORING

These procedures are provided in order to standardise the style in which errors are reported to the user

1) OUT.M (INTEGER, INTEGER)

This procedure outputs, on the currently selected stream, a suitable error message for error class P1 code P2. It is used by compilers, for example, in order to indicate a classified error and then continue to compile the rest of the program.

2) OUT.T (INTEGER, INTEGER)

This procedure outputs, on the currently selected stream, a standard trap diagnostic for error class P1 code P2 (i.e. an error message and other diagnostic information).

10.4 RUNTIME DIAGNOSTICS

The runtime diagnostics of MUSS work in conjunction with the compiler target language MUTL and are thus available for all the programming languages implemented within MUSS. The debugging aids are primarily in source language terms, but machine level information is also provided. Profiling facilities are provided to aid performance analysis of programs.

During a compilation MUTL accumulates information about the data structures of a program and the correspondence between the source and the object code of the program, which is retained in a file. The runtime diagnostics of MUSS use this information to provide diagnostic and profiling facilities. For example, on encountering an error in the execution of a batch program a post mortem is produced in source language terms. This consists of a list of activated procedures and their data structures. Whereas for interactive execution of a program, only the position of the error and the reason is monitored. Runtime diagnostic enquiry commands may then be used to diagnose the error.

The breakpoint and trace facilities of the diagnostic system are invoked by inserting diagnostic checkpoints into the code of a program. For example, to insert a breakpoint in a program the command CHECK.POINT is called specifying the required source text position. When program control reaches the breakpoint the trap procedure entered allows the user to type runtime diagnostic enquiry commands to interrogate his program. The command RETURN returns control to the executing program. The command REMOVE.CHECK.POINT is called when the breakpoint is no longer required. Tracepoints are also inserted and removed by the commands CHECK.POINT and REMOVE.CHECK.POINT respectively. When program control reaches a tracepoint a trace message is output and control is then returned to the executing programs.

Further commands are provided to support the profile analysis of program control. Data are collected by sampling the program control register at fixed intervals, typically once every 50 milliseconds of process execution time. The list of control addresses collected is placed in a file. Analysis of the filed data may then be obtained by calling one of the analysis reporting procedures such as LIST.PROFILE.

The primary diagnostic enquiry commands are LIST.PROCS, EXAMINE

and MODIFY. The names of procedures with frames on the procedure call stack, i.e. the currently activated procedures, are listed by the command LIST.PROCS, while EXAMINE and MODIFY provide for data examination and modification in source language terms.

The secondary diagnostic enquiry commands, FORMAT and SELECT.-DATA, change defaults for the EXAMINE and MODIFY commands. FORMAT varies the manner in which data values are printed, while SELECT.DATA defines the data declarations that the EXAMINE and MODIFY commands check when looking for requested data items. For example, data declarations of textually enclosing procedures can be selected.

As an alternative to interrogating the program state by the enquiry commands EXAMINE, MODIFY and LIST.PROCS, a file containing a source language post mortem may be obtained by the POST.MORTEM command. Standard editing facilities allow the post mortem file to be viewed in a convenient interactive manner.

For system program development three further enquiry commands, namely OUT.STACK, OUT.PROG and OUT.CODE are provided. For a requested area of store OUT.STACK outputs a hexadecimal dump, while OUT.PROG outputs an assembler-like listing. OUT.CODE is similar to OUT.PROG but in addition it gives the correlation between source language statements and machine code.

After interrogating the program state at a break or an error one of the commands RETURN or QUIT is called to either continue or abandon program execution. Instruction tracing may be requested when continuing with program execution.

Although the main use of the diagnostic enquiry commands is on errors and at breakpoints, they may be called from elsewhere. For example, to inspect global data of a library at normal command level.

Several commands require positions in the source of a program or library (source text positions) to be specified. This may be done in one of two ways but in both cases the program or library concerned must have been previously selected as the 'current source' by use of the SELECT.SOURCE command. By default the current program will be the current source.

One way of specifying a source text position is by procedure name and line number within the procedure, for example

SQROOT 10

would define source line number 10 in a procedure called SQROOT. If the line number is omitted the first line of the procedure will be assumed, and if the procedure name is omitted the line number will be taken to be relative to the start of the program.

An alternative way of specifying source text positions is possible when certain program generation tools have been used to generate the source, for example, Flocoder. In the case of Flocoder the flowchart title and box number may be used instead of the procedure name and the line number will be assumed to be relative to the start of the box. Thus the line containing FI in box 6 of figure 11.2.2 might be specified as

SYSINT03.1 6 3

If flowchart names and procedure names are not unique the name will be interpreted as a flowchart name.

In several commands the state of the procedure call stack is relevant. The term dynamic chain refers to an ordered list of procedures that have frames on the call stack, starting with the procedure most recently called. Usually a procedure name identifies uniquely an activation of a procedure, except when there are multiple activations of the same procedure due to recursion. To resolve this anomaly all currently activated procedures are assigned activation identifiers, starting from one, which represents their position down the dynamic chain of procedures. The activation identifier is output by the command LIST.PROCS.

A brief summary of the commands follow, and details of the individual commands are given later in the chapter.

10.4.1 Brief Summary of the Runtime Diagnostic Facilities

The diagnostic facilities are classified as follows:

Break and Tracepoint commands

CHECK.POINT	Define a break or tracepoint at a specified source text position.
CHECK.WAIT	Inhibit action until checkpoint has occurred a further specified number of times.
OUT.CHECK.POINT	List status of the specified checkpoint.
REMOVE.CHECK.POINT	Remove a specified checkpoint.
SELECT.SOURCE	Select library or program as the current 'source'
Profiling commands	
PROFILE	Start profile analysis data capture.
STEER.PROFILE	End profile analysis data capture.
LIST.PROFILE	Produce profile analysis report in source language terms.
LIST.PAGE.PROFILE	Produce profile analysis report in terms of code pages accessed.
Diagnostic Enquiry commands	
POST MORTEM	Produce a post mortem in source language terms in the file specified.
LIST.PROCS	Display Information on procedures currently active.
EXAMINE	Display specified data.
MODIFY	Modify specified data.
FORMAT	Vary the format in which data values are displayed.
SELECT.DATA	Change the data declarations that the EXAMINE and MODIFY commands check when looking for requested data.
OUT.STACK	Output hexadecimal dump of requested area of store.
OUTPROG	Output assembler-like listing of requested area of store.
OUT.CODE	Similar to OUT.PROG but with correlation between source language statements and machine code.
Diagnostic Control commands	
RETURN	Return control from a break or error.
QUIT	Abandon command in which break or error occurred.

10.4.2 Breakpoint Commands

1) CHECK.POINT (INTEGER, ADDR [LOGICAL8], INTEGER)

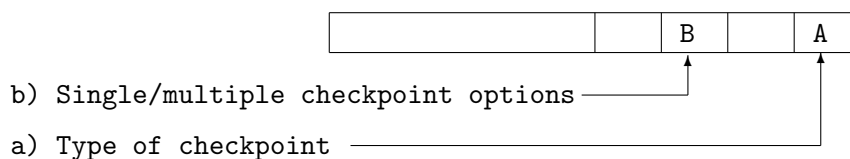
The command inserts a diagnostic ‘check’ into the code at a point whose position is supplied in source language terms or as a byte address. The inserted check can be a ‘break’ point, a ‘trace’ point or a ‘profile’ point. When program control reaches a breakpoint user interaction occurs as if a program error had occurred. When program control reaches a tracepoint, a message is printed and execution of the program then continues. The runtime diagnostic system merely keeps a count of the number of times program control passes a profile point. This count is output by the command OUT.CHECK.POINT.

The position of the checkpoint is given as a source position within the current ‘source’, which is set by SELECT.SOURCE, or as a byte address. The former will be the norm but in some situations it may be necessary to insert a checkpoint in the code identified by its byte address within the code.

The mode parameter P3 states the kind of checkpoint and whether to insert a single or multiple checkpoint.

The multiple checkpoint option inserts a sequence of checkpoints in the program in a systematic way. In this case P2 can either be a procedure name or a source text position. If P2 is not specified then checkpoints are placed throughout the current ‘source’ library. Checkpoints within a procedure or program are placed at the start of each line, or on the first line of embedded procedures. In the latter case P3 also nominates whether checkpoints are placed on all embedded procedures, or only procedures one textual level down. When P2 specifies a source text position it identifies a portion of source text. In this case checkpoints are placed on each line within the portion of source text, or only on the first line of embedded blocks of source. An example of the latter would be to place a checkpoint on each box of a flowchart in a Flocoder program.

P3 is interpreted as follows:



Notes:

a) The value of this 4-bit field is interpreted as:

- 0 – breakpoint
- 1 – tracepoint
- 2 – profile point

b) The value of this 4-bit field is interpreted as:

- 0 – insert a single checkpoint
- 1 – insert a checkpoint on all lines of source specified by P2
- 2 – insert a checkpoint at the first line of source blocks in the source specified by P2
- 3 – insert a checkpoint on all lines of procedure P2
- 4 – insert a checkpoint on the first line of all embedded procedures in procedure P2
- 5 – as 4 except that checkpoints are only inserted procedures one textual level down.

The parameter P1 is normally a number (> 0) by which the checkpoint is referred to in other commands (e.g. `REMOVE.CHECK.POINT` and `OUT.CHECK.POINT`). If P1 is - 1 then a checkpoint number is allocated automatically. The action for other P1 values is as follows:

- 0 – a menu is displayed for the manipulation of checkpoints
- 2 – this validates that checkpoints are still operative; any invalid checkpoints are discarded.

2) `CHECK.WAIT (INTEGER, INTEGER)`

This command inhibits the action of checkpoint P1 until program control has passed the checkpoint a further P2 times.

3) `REMOVE.CHECK.POINT (INTEGER)`

This command removes the checkpoint P1. A value of -1 for P1 means remove all checkpoints and a value of -2 removes all checkpoints from the currently selected source.

4) `OUT.CHECK.POINT (INTEGER)`

This command lists on the current output stream status information for checkpoint P1. For profile points the profile count is also printed. A value of -1 for P1 means print status of all checkpoints.

5) SELECT.SOURCE (ADDR [LOGICAL8])

This command nominates the library P1 as the current ‘source’. All further source text positions are to this library until SELECT.SOURCE is called again. The default for P1 is the current program.

Examples of Checkpoint commands

SELECT.SOURCE ABC	Nominates library ABC as the current ‘source’.
CHECK.POINT 1 “PQ 105”	Set a breakpoint (checkpoint 1) at line 105 of procedure PQ in current ‘source’.
CHECK.POINT 2 “LM 106”	Set a breakpoint at line 106 of procedure LM in the current ‘source’.
CHECK.POINT 3 “RS1.7 3 2”	Set a breakpoint at line 2 of box 3 of flowchart RS1.7 in current ‘source’.
CHECK.POINT 4 “%A0104” 1	Set a tracepoint at control address %A0104.
CHECK.POINT 5 “RS1.7 1” %2D	Set a breakpoint on every box of flowchart RS1.7 in current ‘source’.
REMOVE.CHECK.POINT 1	Remove checkpoint 1
CHECK.WAIT 2 3	Inhibit checkpoint 2 until control passes it three more times.

10.4.3 Profiling Commands

There are two phases of profile analysis, namely data capture and producing a report containing an analysis of the captured data. The commands PROFILE and STEER.PROFILE initiate and terminate data capture respectively, leaving the captured data in a file. If only parts of a program or job are to be profiled then the command STEER.PROFILE, with suitable parameter options enables data capture to be temporarily halted and restarted as required. The procedures LIST.PROFILE and LIST.PAGE.PROFILE produce an analysis of the captured data on a procedure and page basis respectively.

1) PROFILE (ADDR [LOGICAL8], INTEGER, INTEGER32)

This command is called to initiate the data capture phase of profiling. The name of the file in which captured data is placed is specified by P1. The sampling interval of the program control (P2) is specified in milliseconds, and the maximum number of samples is specified by P3.

2) STEER.PROFILE (INTEGER)

This command steers the data capture phase of profiling. P1 is interpreted as follows:

- 0 – terminate data capture
- 1 – temporarily suppress data capture
- 2 – restart data capture.

3) LIST.PROFILE (ADDR [LOGICAL8], ADDR [LOGICAL8], ADDR [LOGICAL8])

This command produces an analysis in source language terms of the data captured in file P1 on the destination document P2. If P3 is not specified the analysis is at a procedural level. Otherwise P3 is a procedure name, in which case, a comprehensive profile analysis of control within procedure P3 is given.

4) LIST.PAGE.PROFILE (ADDR [LOGICAL8], ADDR [LOGICAL8], INTEGER)

This command produces an analysis in terms of code pages accessed of the data captured in file P1 on the destination document P2. P3 specifies the page size in bytes.

Examples of Profiling commands

```
PROFILE DUMP 25
RUN MYPROG
STEER.PROFILE
```

The above commands run program MYPROG and the diagnostic system samples the program counter every 25 milliseconds of process execution time leaving the captured data in file DUMP.

LIST PROFILE DUMP LPT* This produces on the lineprinter a report

giving a profile analysis of the procedures in which control was sampled.

LIST.PAGE.PROFILE DUMP LPT* %400 This produces on the lineprinter a report having a profile analysis of the code pages of size 1 K bytes in which control was sampled.

10.4.4 Diagnostic Enquiry Commands

1) POST.MORTEM (ADDR [LOGICAL8])

This command produces a post mortem in source language terms on the destination document P1.

2) LIST.PROCS (INTEGER, INTEGER, ADDR [LOGICAL8])

This command scans the dynamic chain of active procedures, and lists the name, the activation identifier, and the position of program control in these procedures. P2 specifies for how many procedures information is printed, while P3 specifies the starting point of the scan in the dynamic chain. This starting point may be given as a procedure name or an activation identifier, the default is the first procedure in the dynamic chain. Normally only source language information is printed, but if P1 is non-zero lower level information such as the address in hexadecimal of program control within procedures is also printed.

3) EXAMINE (ADDR [LOGICAL8], ADDR [LOGICAL8], ADDR [LOGICAL8], INTEGER) LOGICAL64

This command displays the contents of 'data' requested by P1. Data values are printed in a format specified by P2. Finally P3 and P4 specify the data declarations to check in locating the requested data.

The 'data' requested can be one of the following:

- a) O – all variables
- b) *P – parameters
- c) *R – all registers
- d) *name – a register
- e) A MUSL-like data operand.

‘O’ means print all data items located by the data search, typically this option is used to print all the variables of a procedure. ‘*P’ means print parameter variables located by the data search. ‘*R’ means print all machine registers, and ‘sname’ means print the named register. Otherwise P1 is taken as a data variable or the desired component of a data variable. This is specified in a similar manner to a data variable operand of an expression in the MUSL language, but there are some differences. The main difference is the form of a subscript expression. It can be a constant, as in MUSL, in which case just a single element of the vector is displayed. Alternatively, a number of consecutive elements can be displayed by specifying the lower and upper limits of the required slice. For example,

EXAMINE “MODE OF STREAM [3:5]”

will display the MODE field of elements 3, 4 and 5 of the vector STREAM. In the example the enclosing quotes are needed by the command interpreter to delimit the parameter. Either or both of the slice limits may be omitted. If there is no lower limit specified then the slice starts at element 0, and if there is no upper limit the last element of the slice is the last element of the vector. Finally ‘[]’ and ‘[:]’ are equivalent and mean the slice is the complete vector.

Displaying data at known addresses is also provided for. To do this the location and the type of the data object is specified. For example,

EXAMINE “%A0422:\$IN16”

will print the value of the 16-bit integer at location %A0422. The type can be a scalar or a vector of basic or user defined type.

When no type is given the type most recently specified in a data enquiry command is used. The detailed syntax of an operand, expressed in the BNF-like notation of Chapter 9, is as follows:

```

<OPERAND>          ::= [(<FIELDS>!<NIL>)<DATA.STRUCTURE>
                        [<MODIFIER><NIL>]]
<DATA STRUCTURE> ::= [<DATA NAME>!<VIRTUAL.ADDR>
                        [:<TYPE>!<NIL>]]
<MODIFIER>         ::= <ELEMENT>[↑!<NIL>]↑! [<ELEMENT>!<NIL>]
<ELEMENT>          ::= <[>(<INTEGER>!(<INTEGER>!<NIL>))>:

```

```

                                [<INTEGER>!<NIL>!<NIL>]!<]>
<FIELDS>                      ::= <FIELD>OF[<FIELDS>!<NIL>]
<FIELD>                       ::= <FIELD NAME><MODIFIER>

```

<TYPE> is specified as in a variable declaration of the MUSL language (see Chapter 9)

The format control parameter P2 is described in the FORMAT command below. The parameters P3 and P4 are described in the SELECT DATA command below.

The result of EXAMINE returns either the value or the address of the ‘data’ depending on the ‘format’ specified.

4) MODIFY (ADDR [LOGICAL8], ADDR [LOGICAL8], ADDR [LOGICAL8], INTEGER)

This command displays the contents of ‘data’ requested by P1 as in EXAMINE but prompts after each value is printed. In response a new value may be given or a new position selected in the data structure. For example, if the current data item is a pointer, a new position obtained by dereferencing the pointer may be chosen, thus allowing linked list to be traversed. The parameters are interpreted as in the EXAMINE command.

Data variables in a program are normally thought of as having a hierarchical structure, where the type of all components except those at the lowest level in the hierarchy are regarded as user defined type. However data may also be considered as a logical sequence of simple components where each component is a scalar of basic type or a vector of basic type elements. The MODIFY command views data structures in this way and prints the next value in the sequence and awaits one of the following responses:

VALUE – New value. Two values are needed for a bounded pointer, the first value is the new bound and the second the new address.
 NEWLINE – Print the next value.
 @ – terminate modification.

5) FORMAT (ADDR [LOGICAL8], ADDR [LOGICAL8])

This command varies the default formats for the data values displayed in the EXAMINE and MODIFY commands. The formatting rules are given

by P1, while P2 specifies to which data types they apply. P1 is a string composed of the following characters:

A	print address of value
B	print as a binary number
C	print as a character string
H	print as a hexadecimal number
I	print as an integer number
O	print as an octal number
R	print as a real number
T	print the type of the value
V	return value in EXAMINE result
X	return address in EXAMINE result.

If P2 is not specified, values of all data types are printed in the requested manner. P2 specifies a list of data types separated by commas. These data types are the numeric types as defined in the MUSL language, but the additional types are recognised:

*I	all integer types
*L	all logical types
*R	all real types
*A	all pointer (ADDR) types

If P1 is not specified then the original defaults are selected for the types P2.

6) SELECT.DATA (ADDR [LOGICALS], INTEGER)

This command sets the defaults for the data declarations that the EXAMINE and MODIFY commands check when locating requested 'data'. P1 specifies a group of data declarations and the options are as follows:

- a) A procedure name or an activation number nominates the data declarations of a currently active procedure. P2 specifies whether related declarations are required, zero means no other declarations, one means nominate also the declaration of textually enclosing active procedures, and two means nominate also the declarations of procedures after PI in the dynamic chain of activated procedures.

- b) A library name. This means search global data declarations of the module P2. If P2 is zero then all modules of the library are searched. Modules of a library are numbered from one upwards. If library names and procedure names are not unique P1 will be interpreted as a library name.

The initial default for SELECT.DATA is the data declarations of procedures in the dynamic chain of activated procedures. Calling SELECT.DATA with P1 unspecified resets to these defaults.

7) OUT.STACK (ADDR, ADDR)

This produces a hexadecimal dump on the currently selected output stream of the area between byte addresses P1 and P2. The format of the dump is the hexadecimal address of the start of each line, followed by four integer sized words in hexadecimal followed by the character representation of the bytes. Any number of consecutive identical lines are replaced by one copy of the line followed by a single blank line. If P2 is zero then a suitable default is taken.

8) OUT.O.STACK (ADDR, ADDR)

This produces an octal dump on the currently selected output stream of the area between byte addresses P1 and P2. The format of the dump is the octal address of the start of each line, followed by four integer sized words in octal followed by the character representation of the bytes. Any number of consecutive identical lines are replaced by one copy of the line followed by a single blank line. If P2 is zero then a suitable default is taken.

9) OUT.PROGn (INTEGER, ADDR, ADDR)

This produces on the currently selected output stream an assembler-like listing of code between P2 and P3 of segment P1. The 'n' in the command name selects the appropriate machine as follows:

3 VAX
5 MC68000

If the abbreviated command OP is called code for the current machine is output. P2 and P3 are specified in bytes. If P3 is zero then a suitable default is taken.

10) OUT.CODE (ADDR [LOGICAL8], ADDR [LOGICAL8])

This produces an assembler-like listing of code between the source text positions P1 and P2 on the currently selected output stream. P1 and P2 are specified as the source text parameter P2 of CHECK.POINT. Alternatively, P1 and P2 may be specified as hexadecimal byte addresses. The listing produced gives the correlation between source text and object code. If P2 is not specified then a suitable default is taken. If P1 is not specified then the code where the runtime occurred is listed.

Examples of Diagnostic Enquiry Commands

EXAMINE	Display all variables found
E	Display all parameters found
E I	Display variable I
E I 0 EVAL	Display variable I in procedure EVAL
E I 0 4	Display variable I in procedure whose activation number is 4, i.e. fourth most recently activated procedure
E "MODE OF STREAM [4]"	Display MODE field of fourth element of vector STREAM
E "MODE OF STREAM []"	Display MODE field of all elements of vector STREAM
E STREAM	Display vector STREAM
E "%A2456:LO16[32]"	Display vector containing 32 elements of 16-bit unsigned integer type located at address %A2456
E "%440000:LIBTYPE"	Display variable of type LIBTYPE located at address %440000
E "PCNT OF %440000:LIBTYPE"	Display PCNT field of variable in previous example
FORMAT HR \$RE32,\$L032	All 32-bit real values and 32-bit logical values are to be printed in hexadecimal and floating point format
F HR *R	All real values to be printed in hexadecimal and floating point format
E XVAL HRB	Display XVAL in hexadecimal, floating point and binary format
E *R	Display all machine registers
E *D2	Display machine register D2
E: *D2 IHC	Display machine register 02 in integer, hexadecimal, and character format
LIST PROCS	Display all activated procedures
LP 0 3	Display first three procedures in the dynamic chain.
OUT.STACK %10000 %10100	Output hexadecimal dump of 256 bytes starting at address %10000.
OUT.CODE.TEST	List the code at the start of procedure TEST.
OUT.CODE "TEST 3"	List the code starting at box 3 of procedure TEST.
OUT. PROG 5 %200 %300	List 256 bytes of code starting at byte %200 in segment 5

10.4.5 Diagnostic Control Command

1) RETURN (INTEGER)

This command returns control from a break or an error. P1 specifies how to return and the options are as follows:

- a) 0 – Continue execution of the program or command in which break or error occurred.
- b) > 0 – Continue execution but monitor the next P1 instructions executed then break again.

2) QUIT ()

This command abandons the command or program in which the break or error occurred.

10.5 LOW LEVEL RUNTIME DIAGNOSTIC COMMANDS

The INIT RD commands is not normally called by the user, but by the system to initialise runtime diagnostics for a process.

1) INIT.RD (INTEGER)

This command initialises the runtime diagnostic system. The action on program errors is selected by P1. The following options are available:

- 0 Output a post mortem on the monitoring stream and stop the process.
This is the norm for a batch job
- 1 Monitor the cause of the error on the monitoring stream and accept further program control commands from stream zero. This is the norm for an interactive job.
- 2 Output a post mortem on the monitoring stream, abandon the command or program in which the error occurred, and return control to the command processor.
- 3 As 2 but only the reason for the error is monitored.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 11

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

11 DOCUMENTATION AIDS

The Documentation aids are of two kinds: word processing and diagrammatic. Most users should find the first set of general interest while the second set relates to the particular programming methodology used in producing MUSS. This programming methodology is intended to have wider applicability than the development and maintenance of MUSS, but is based on a definite viewpoint which may only be appropriate in certain cases.

11.1 WORD PROCESSING

These facilities are provided mainly by the procedure TXT, but a spelling checking aid, SPELL, and a word counting facility, WCOUNT, augment this, and the standard editing and file facilities described in earlier chapters provide the means for manipulating the files. There are also some output procedures for printing a dictionary or index. TXT takes an encoded description of a document, and produces the required layout. It contains a faculty to interface with other layout procedures, for example for generating tables, formulae and diagrams.

11.1.1 The Text Facility

The procedure TXT copies text from an input stream generating a layout suited to a specified type of printing device. It allows the user to embed “warning sequences” in the text which control the layout. Four basic device types are provided for, corresponding to Diablo type daisy wheel printers, Santex Corporation Variflex printers, Laser printers and ordinary printers or terminals. All printers are assumed to have a full visual character set but the daisy wheel. Santex and Laser printers are also assumed to have:

- a variable character size
- a linefeed upwards
- a variable sized linefeed
- an underline facility
- and variable fonts (Santex and Laser printers only).

Normally the page and line layout is determined by the TXT procedure, and the user is required only to provide the actual text and to indicate paragraph and section boundaries. This manual has been produced using

TXT and serves as an example of the facility. The formal specification of TXT is:

TXT (ADDR [LOGICAL8], ADDR [LOGICAL8], LOGICAL64.
ADDR [LOGICAL8], ADDR [LOGICAL8])

where P1 specifies the input file
P2 specifies the output file
P3 = 0 ordinary lineprinter
 = DBL diablo printer
 = LPT line printer (as 0)
 = SAN Santex printer
 = LAS Laser printer
P4 = specifies a contents file
P5 = specifies an index file.

If P4 and P5 are omitted the contents and index information are suppressed. The contents file when produced will contain entries for all Chapter and Section headlines (i.e. @R, @S1, @S2, @S3 described below) in TXT format with chapter and page numbers added.

The index file will be an unsorted list of words with chapter and page numbers. It is normally expected that a document will be produced a Chapter at a time, and that the index files will be edited together and listed using the facilities described in [11.1.7](#).

It is the warning sequences and the actions they trigger that characterise the system. These are presented in three groups as the basic facilities, the layout control facilities and the special purpose layout facilities. However, most warning sequences normally start with the symbol '@'. The character immediately following the '@' defines the required action, some of which may require that some parameters follow.

11.1.2 Basic Facilities of Text

The basic mode of operation of TXT is that it copies words from the input to the output, starting newlines (and newpages) as necessary to avoid splitting words across lines and spacing the words across a line to right justify them. A 'word' in this context is any sequence of characters in the input separated by space and/or newline characters and/or warning sequences. Warning

sequences inserted in the input can modify this basic action as follows.

@ followed by a newline symbol. This causes a newline to be inserted and it inhibits the right justification of the current line.

@ followed by B. This signifies the start of a new paragraph. The current line is terminated without right justification, blank lines are inserted to separate the paragraphs and spaces are inserted to indent the new paragraph.

@ followed by R indicates the start of a new chapter. It should be followed by a string giving the chapter name. If a chapter number has been specified (see @V 11.1.3) it will be output prior to the page number, for format and style see the page numbering at the top of this page. The chapter string also appears as the central page heading on all righthand pages if the 'double-sided' heading option is selected (see @V 11.1.3). The position of the chapter heading will be determined relative to current margin settings. An entry will also be generated in the contents file.

@ followed by S1, S2 or S3. This indicates the start of a new chapter or section. The action is similar to @B except one more blank line is normally inserted and the following line is not indented. In fact the following line in the input is copied without any layout change to the output because it is assumed to be a heading, apart from the line being made bold and requires an @G to switch the bolding off. This facility is described later. There is no distinction between S1, S2, S3 from the point of view of the formatted text. However, they cause different degrees of indentation and separation in the contents file. S1 should be used for chapter headings (as an alternative to @R), S2 for sections and S3 for subsections.

@ followed by P. This causes a newpage to be started and the last line of the previous page will not be right justified.

@ followed by G. This provides a bolding facility on devices which support it, otherwise it is ignored. Normally it works as a toggle but the bold state is automatically switched on by the OS facilities as described above.

@ followed by O. This is again concerned with underlining. It ‘toggles’ an ‘underline switch’ The first use of it will ‘turn on’ the underlining of all following characters until a second use of it turns the underlining off. It also is reset after predictable newlines.

@ followed by K. This is used for highlighting changes by means of margin bars. The first occurrence of @K causes a switch to be set. Whilst this switch is set all lines that follow will be preceded by ‘—’ in the margin. A second occurrence of @K resets the switch and the lines which follow will not be highlighted, unless a further @K occurs to set the switch again.

@ followed by X. In addition to ‘@’ four other symbols, namely ‘%’, ‘{’, ‘}’ and ‘_’ have special significance as described in [11.1.3](#). If text is to be processed which uses these symbols normally, other symbols can be associated with their special functions, by means of the @X command. After @X should follow a list of non-blank symbol pairs, in which the first symbol is one of @, %, {, } or _ and the second is the symbol that is to take its place.

@ followed by one or two digit numbers. These sequences are concerned with font selection. The number specifies a font number and should be in the range 1–20 in order to explicitly specify a font, and the font change will take place on the following character as long as this character is not a single digit number. Each font has fixed character sizes which relate to a compiled in default. New fonts may be defined as described below.

@ followed by L. This command, for defining the characteristics of a font is only applicable to the Santex and Laser printer and will be ignored in other cases. The Santex printer is now regarded as obsolete and therefore the definition of fonts for this printer is not described in this Manual. In the case of the Laser printer, if a user defined font is selected for text output with a different size and direction the command @L must be followed by the new font style number, the predefined font number, the intercharacter gap in pixels, the width scaling factor, the height scaling factor, and the next two parameters are normally set at zero unless a height scaling factor of the width and a width scaling factor of the height is required. Character sizes are specified in terms of 300ths of an inch. (See [12.6](#)).

@ followed by W. This command should be followed by a string terminating with @. It defines a footnote which will appear below the last line of the current page, highlighted by an asterisk.

@ followed by Y. This command should be followed by a string terminating with either newline or @, newline. It defines a page heading, which will appear centrally at the top of a page. If the single-sided layout option has been selected (see @V 11.1.3) it will appear at the top of all pages apart from the first. If double-sided it will appear on only left-hand pages and the right-hand pages will have the current chapter title at the top. The current position of the heading is determined relative to the margin settings.

@ followed by F signifies the end of the text file. In order to avoid an entry to the input ended trap of the operating system '@F' should be placed at the end of all files to be processed by TXT. If it is omitted the output text will normally still be produced, and its main use is when the text is followed by other information in the same file, for example, flowchart encodings.

11.1.3 Layout Control Facilities (@V)

Here we are concerned mainly with centring, tabulating and indenting text, and also with superscripts and subscripts. Some of the effects can be obtained by changing the parameters that determine the basic layout, and it is convenient to start with this facility. An @ followed by V provides the warning sequence that causes parameters to be changed. Up to sixteen different parameter values may be specified, so the @V should be followed by an arbitrary list of pairs of integers all on one line separated by spaces. Each integer pair gives a parameter number and its new value. The parameter numbers have the following significance, with their default values given in brackets.

- 1 position of L.H. margin (0)
- 2 position of R.H. margin (0)
- 3 number of blank lines between paragraphs (2)
- 4 number of spaces to indent at the start of a paragraph (3)
- 5 number of blank lines between sections (3)
- 6 number of blank lines at the head of a page (3)
- 7 number of lines on a page (DBL, LPT=60; SAN=57; LAS=67)

- 8 chapter number (0)
- 9 page number (1)
- 10 single/double sided (0 – single sided)
- 11 page heading font (0)
- 12 chapter heading font (0)
- 13 number of blank lines before the chapter heading (5)
- 14 number of blank lines after the chapter heading (10)
- 15 interline gap (1)
- 16 interline gap (2).

Four of the facilities require amplification because they permit special encodings. If a negative number is given for parameter 4 the first paragraph of each new section will not be indented. The other paragraphs will be indented by the absolute value of the number. If a negative number is given for parameter 5, each section heading will be placed on a newpage. If a zero is given for parameter 8, the chapter number will not appear before chapter headings, neither will it appear as part of the page number. If a negative number is given as the page number, parameter 9, pages will not be numbered at all.

Parameter 10 allows control over page headings. A parameter value of 0 selects single-sided, while a value of 1 selects double-sided layout. If single-sided is selected the heading string defined by @Y will appear centrally at the top of each page. Double-sided documents have the heading string on the left-hand page (even numbered) and the current chapter title on the right-hand page (odd numbered).

Any line of input can be centred on a line, this can be achieved by preceding the input with @M. The previous line is terminated without right justification. The position of the first character on a centred line is remembered and any subsequent line of input can be similarly positioned by preceding it by @N. Again the line previous to the @N line will be terminated without right justification. In both cases the output line will correspond exactly with the input line apart from being right shifted to achieve the centralisation effect.

Tabulation is normally provided by the ‘%’ character. Its action is defined by the @T sequence. This serves a dual function, it defines a character which is subsequently to be treated as a ‘tab’ character, normally ‘%’, and it defines positions across the page for ‘tab stops’. Therefore the @T is assumed to

be followed first by a single character which becomes the tab character and then a list of integers which define the positions of tab stops counting from the L.H. margin. Whenever the tab character is used the output line will be space filled up to the next tab stop. If the output line is subsequently right justified, the extra spacing will occur only to the right of the last space inserted as a result of using a tab symbol. Obviously even this can be avoided by ending the source line with @ newline.

Indentation is provided by a mechanism similar to the tabulation facility. The warning sequence is @I, which is similar to % but with some very significant differences. Like 'tab', when 'indent' is used the line is space filled to the next tab position. The difference is that the last indented position is remembered and if a newline is automatically generated in the output due to the current line becoming full, the next line is automatically indented to the same extent. In the case of tabulation, the next line would commence back at the margin. Any user forced newline by means of @newline, @B etc., cancels the indentation.

Providing the output device has the facility of fractional linefeeds up and down, superscripts and subscripts are obtained by surrounding the appropriate character string with the symbols '{' meaning 'shift half a line up the page' and '}' meaning 'shift half a line down'.

Backspacing is provided by the '_' character. The character immediately following the '_' is the character that is to be backspaced to the previous position, for example =_/ will result in ≠.

11.1.4 Special Layout Effects

The special layout effects are concerned mainly with diagrams and tables contained within the text, and with special forms of document.

In the case of diagrams which are to be 'glued into place', the requirement is to leave space at an appropriate place in the text for a diagram that will be produced by other means. The warning sequence for immediate creation of space is @D which should be followed by an integer giving its size in lines. If there is not enough room on the current page a newpage will be started and the requested space will be left blank at the head of it. A further alternative is provided by @E, which leaves the requested amount of space on the current page if possible, otherwise space is left at the top of the next

page while the current page is still used for subsequent text.

Table layouts and diagrams generated by explicit statements may also need to be positioned. For example, when a table is generated using the tabulation and indentation facilities, it may be desirable to prevent it being split across pages. The simplest need is met by @Q which is followed by an integer specifying the number of lines required by a following table. If the specified number of blank lines do not remain on the current page, a newpage is forced otherwise the @Q has no effect. A variant of this is @U which in addition to operating like @Q suppresses all layout changes for the specified number of lines. A second requirement is met by two warning sequences @A and @Z. @A may appear in front of any sequence of text, but most often a table encoding, and it should be followed by an integer giving the amount of space in lines that the text will occupy in the output. The given text whose end should be marked with @Z will be output immediately if there is room on the current page, otherwise it will be output at the head of the next page after subsequent text has been output to complete the current page.

Another way in which tables and diagrams can be generated explicitly by in-line commands is by calling another procedure from TXT using the @ command. This may be followed by any job control command. In particular. If a flowchart is to be inserted it might be required to call the DRAW procedure described in [11.2.7](#).

Finally there is a facility @H which causes the line of text following it to be output in large letters. The style is device dependent and simple character devices will give

```
#      # #####      ##      #####      #####      #      #      ####
#      # #          # # #      #      #      ##      # #      #
##### #####      #      # #      #      #      # #      # #
#      # #          ##### #      #      #      # #      # #      ###
#      # #          #      # #      #      #      #      ##      # #
#      # #####      #      # #####      #####      #      #      ####
```

11.1.5 Summary of Text Commands

- @A displaced text header
- @B paragraph start
- @D leave space for diagram
- @E leave displaced space
- @F end of document

@G toggle the 'bold' switch
@H heading
@I indentation
@J add to index
@K toggle highlighting by margin bars
@L define new font
@M line centreing
@N indentation following line centreing
@O toggle the underline switch
@P newpage
@Q force newpage for large tables
@R chapter heading
@S section start
@T set tabulation stops
@U user formatted diagram
@V reset layout control parameters
@W footnote
@X change warning characters
@Y alter page heading
@Z end displaced text
@* call job control command
@newline print newline
@@ print @
@<decimal digit> select font

11.1.6 The Spell facility

This procedure is simply an aid to checking the consistency of spelling in a text file such as the TXT procedure might generate. It requires two parameters which are the file to be checked and a dictionary. All words in the text file are compared with words in the dictionary and if a match is not found the word is noted. A word in both the dictionary and the text files is any sequence of alphabetic characters between spaces and/or newlines. The list of words whose spelling is not found in the dictionary appear as the current file. Obviously this file can be printed, edited or added to the dictionary as appropriate. A suitable print procedure is given in [11.1.7](#).

SPELL (ADDR [LOGICAL8], ADDR [LOGICAL8])

P1 is a dictionary file name

P2 is a text file name.

11.1.7 Listing Facilities

Three special listing procedures are provided. Two of these allow dictionaries and document index to be sorted and listed. The third allows a complete module composing textual documentation and flowcharts to be TXT'ed and drawn in both English and MUSL on the lineprinter.

1) LIST.DICT (ADDR [LOGICAL8], ADDR [LOGICAL8])

This procedure sorts the dictionary contained in the file specified by P1 and outputs the result to P2. The output has four words per line sorted into alphabetical order with a blank line separating words with a different initial letter

2) LIST.INDEX (ADDR [LOGICAL8], ADDR [LOGICAL8])

This procedure lists the index contained in P1 on stream P2. The output is sorted and formatted. An example of the format produced is the index for this Manual.

3) LIST.MOD (ADDR [LOGICAL8])

This procedure firstly uses TXT to output to the lineprinter the descriptive text at the start of the file P1 and follows this by the flowcharts drawn at all levels using FLOCODER as described below.

11.2 FLOCODER

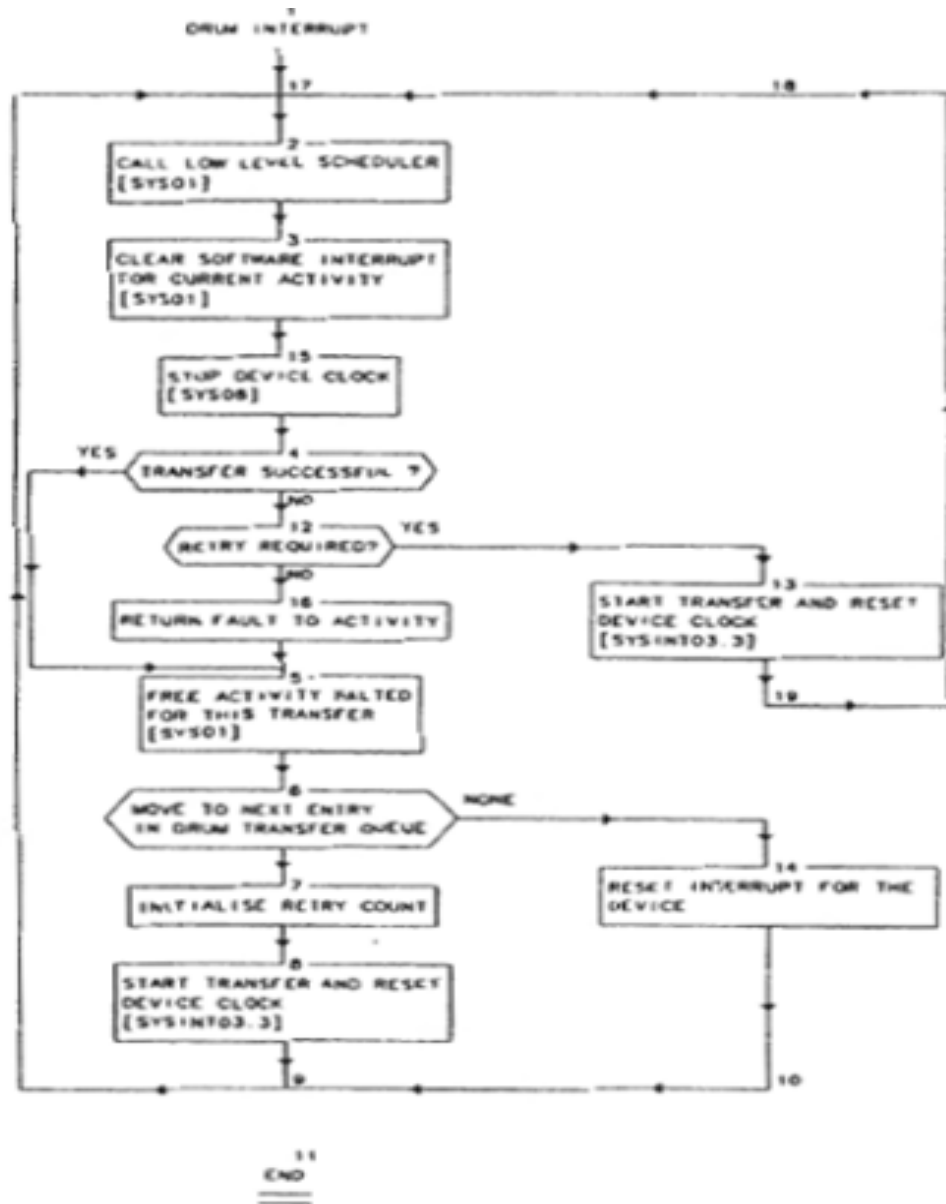
Flocoder is a system for designing, documenting and generating programs using flowcharts. A file of flowchart descriptions is created, from which the charts may be drawn on any suitable output device, such as a lineprinter or plotter. The chart descriptions may of course be edited, and the charts re-drawn as necessary.

To enable Flocoder to generate or display the required program, the user provides a 'translation' for each box. If the action required in a box is simple, it will translate into a sequence of statements in a programming language; if it is complex, the translation may reference other flowcharts. In this way a hierarchy of flowcharts is created to represent the program. In fact, several translations can be given for each box. The first would normally be an English statement describing the logical function of the box and would

be for display purposes only. The programming language translations for each of the required languages would be added later. In effect the Flocoder system comprises a language for describing flowcharts and two procedures for processing this language. One of these, 'DRAW' will draw the flowcharts. The other 'FLIP', will form a linear program by correctly ordering the boxes and adding labels and 'goto's as necessary, although the latter representation is only seen by compilers.

The syntax of the Flocoder input language is simple and straightforward. Each statement in a chart description begins with the symbol '@' as the first character of a line, followed by a keyword, and continues until the start of the next statement. The keywords can be abbreviated to single letters since they are recognised by the first letter only; after that, all characters up to the next space or decimal digit are ignored. A complete chart description consists of

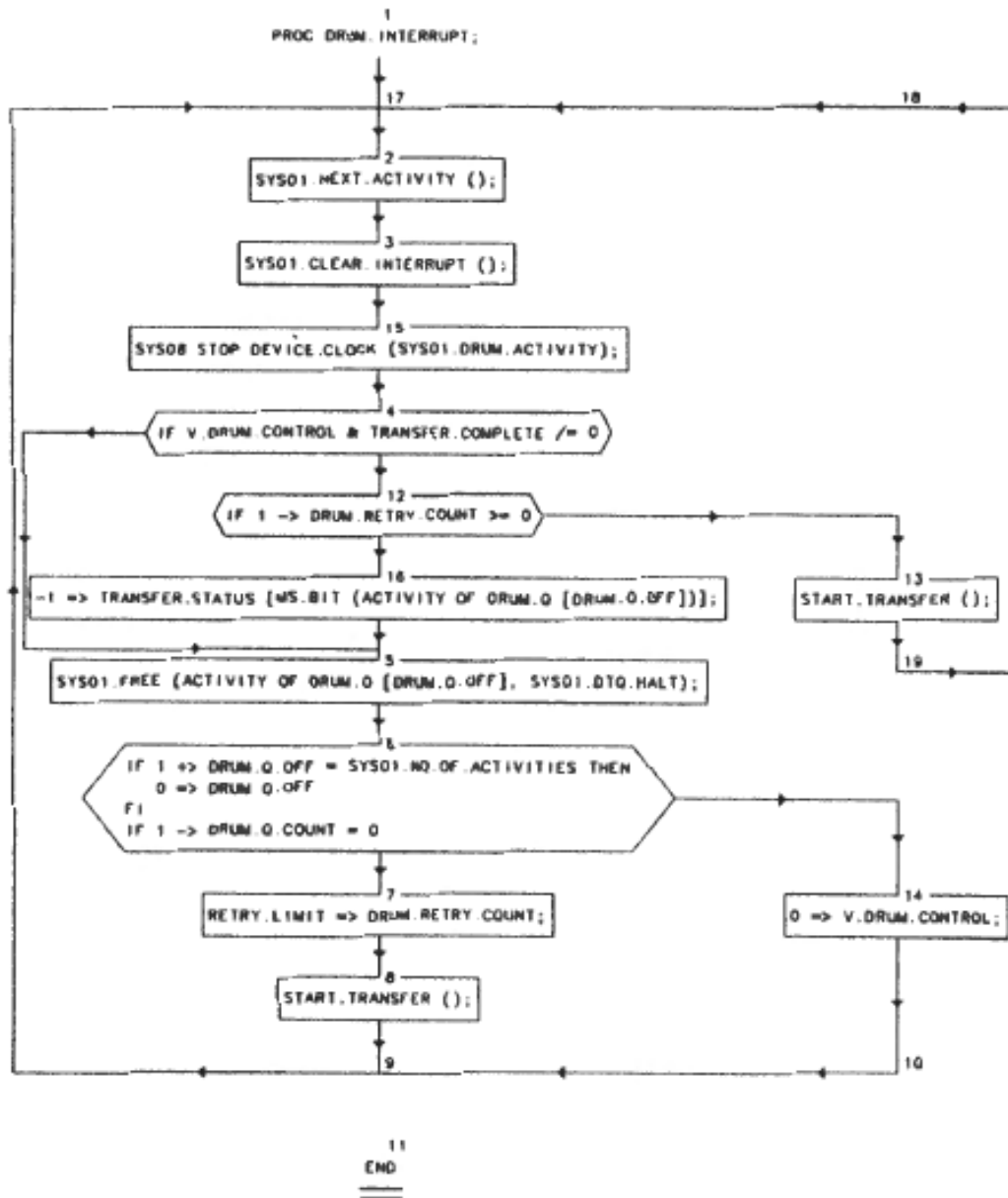
- A TITLE statement
- One or more COLUMN statements
- Zero or more ROW statements
- Zero or more FLOW statements
- One or more BOX statements
- An END statement.



23/05/86

SYSINT03.1(1,7) - 0

Figure 11.1



23/05/86

SYSINT03.1(1,7) - 1

Figure 11.2

These individual statements are described below and are illustrated by examples taken from the encoding of the diagrams in figures 11.2.1 and 11.2.2 above.

11.2.1 The TITLE Statement

Example:

```
@TITLE SYSINT03.1(1,7)
```

The TITLE statement indicates the start of a new chart and gives a title which serves two functions. First, it appears on the flowchart whenever it is drawn and thus serves to identify the drawing. Second, it is used in cross references within the code. A chart title may consist of any sequence of characters, terminated by a newline symbol and may optionally include version and generation numbers enclosed in round brackets. By convention, the characters are usually chosen so as to provide an index into the software. Thus the title in the example above is the first chart in the file SYS03. The INT has been added as a reminder that it is a procedure which runs in interrupt mode.

11.2.2 The COL Statement

Examples:

```
@COL 1S-17N-2R-3R-15R-4T-12T-16R-5R-6T-7R-8R-9N-11F  
@COL 18N-13R-19N-14R-10N
```

The column statements provide, for each box, a numeric identifier in the range 1–63, the type (shape) of the box, and the position of the box on the flowchart. A chart may contain up to eight columns. The first column statement describes the leftmost column, and the last one the rightmost column. If there is more than one box in a column, the first one specified is the highest in the column and the last one the lowest, etc. The box types, which follow the box numbers, consist of single letters with the following meanings

Letter Meaning

- | | |
|---|--------------------------------------|
| A | Annotation box (no outline) |
| C | Circle box (used for external flows) |
| F | Finish box (double underlined) |

N	Null box (a point)
R	Rectangle box
S	Start box (no outline)
T	Test box

The most commonly used box types are S, F, R and T, although the example given also makes use of NULL boxes to route flow lines. By convention the circle box is used for explicit labels and will rarely be required.

11.2.3 The ROW Statement

Examples:

```
@ROW 17-18
@ROW 13-16
@ROW 7-14
@ROW 9-10
```

Each row statement gives a list of boxes to be horizontally aligned. The ordering of the box numbers in the row statements has no significance. Normally the boxes within a column are placed a minimum distance apart, and may be imagined as being connected to the box above (if any), or to the top of the diagram (in the case of the first box of a column) by invisible elastic. This means that boxes tend to be as high in their columns as possible. The effect of the ROW statement is to force horizontal alignment by 'stretching the elastic'.

11.2.4 The FLOW Statement

Examples:

```
@FLOW 1-17-2-3-15-4YES-5-6-7-8-9-17
@FLOW 4NO-12YES-13-19-18-17
@FLOW 12NO-16-5
@FLOW 6NONE-14-10-9
```

These statements specify the logical interconnections of the boxes. Text which is to appear at the point where a flowline leaves a box may also be specified in the flow statements. Any string of characters excluding newline and hyphen is allowed. Except for test boxes, which may have two, there should be not more than one flowline leaving each box. Of course, the finish box will have no flow out.

11.2.5 The BOX Statement

Examples:

```
@BOX 1.0
DRUM INTERRUPT
@BOX 2.0
CALL LOW LEVEL SCHEDULER
[SYS01]
@BOX 1.1
PROC DRUM.INTERRUPT;
@BOX 2.1
SYS01.NEXT.ACTIVITY ();
```

These statements specify the text contained within each box, which may consist of any number of lines up to the start of the next statement. Several ‘translation levels’ may be defined for each box, corresponding to translations in several different languages. The example above gives translations in English at level 0 and the system design language (MUSL) at level 1. When the charts are drawn any translation level can be selected for display in the boxes. The first chart was produced by specifying level 0 (English) and the second by specifying level 1 (MUSL). Similarly, the procedure FLIP can be instructed to generate code from any translation level.

Flowchart cross-references may be inserted in the code by giving the name of the chart to be included, preceded by the character # at the start of a line. This is not shown in the above example, however the file from which it is taken contains a line

```
#SYSINT03.1
```

at the point where the code for the DRUM.INTERRUPT procedure is to be substituted.

In some languages (e.g. Pascal) it is necessary to declare the labels which are used. Therefore a variant of the cross reference notation is provided for this purpose. If #:chart name is used it will be replaced by the list of label names used in the given chart separated by commas.

11.2.6 The END Statement

Example:

```
@END
```

This statement terminates the description of a flowchart.

11.2.7 Flocoder Procedure Specifications

There are six procedures in the Flocoder system as follows

1) FLIP (ADDR [LOGICAL8], INTEGER, INTEGER, ADDR [LOGICAL8], ADDR [LOGICAL8], ADDR [LOGICAL8])

This procedure converts the flowchart specifications on the specified input document, into a linear program corresponding to a particular level of coding. The next item on the current input stream should be the title of the chart to be converted. The linear program which is generated becomes the current file.

In order to generate a linear program FLIP has to place the code form of the boxes in the correct order, and introduce some labels and 'goto's. Since the style of these insertions varies between programming languages there are parameters which specify the required style. Normally the conditional 'goto' string is generated after each test box and other 'goto's are only inserted when boxes connected by flow cannot be placed in sequence. In some cases the conditional 'goto' needs to be inside the code of a test box, therefore a notation is provided (@>>) to allow an explicit indication of placement of conditional 'goto's. If this option is used the automatic placement of a conditional 'goto' at the end of a test box is suppressed. An example of its use is

```
@BOX 45.1
FOR I < 100 DO
    IF LIST [I] = 0 @>>
OD
```

where box 45 is assumed to be a test box.

Additional information is added to the linear program to show it's correspondence with the original flowchart. Text of the following form is added:

|A flowchart name indicates the start of a flowchart

|B box number indicates the start of a box

Z box number	indicates the end of the previous box and the start of the next box
	indicates the end of the last box of a flowchart.

P1 – name of the input file (or stream, see Chapter 3). If this is left unspecified, the current file is used. If no current file is defined, the currently selected input stream is used.

P2 – the required coding level

P3 – This is the MOD LEVEL which specifies the level of code which should supersede the required code level (P1). Thus, if a BOX exists for level P2 and P3 the BOX at level P3 will be used. This allows alternative versions of code for boxes to allow for new developments and alternatives.

P4 – a string giving the form of labels required by the intended language.

P5 – a string giving the form of ‘goto’s required.

P6 – a string giving the form of conditional ‘goto’s required.

In the strings P4, P5, P6, a unique numeric identifier is inserted immediately prior to the last character. If any of P4–P6 are zero, the following defaults, which are suitable for MUSL programs, are assumed:

P4 – L: giving labels of the form Ldddddd;

P5 – – > L; giving gotos of the form – > Ldddddd;

P6 – , – > L; giving conditionals of the form , – > Ldddddd;

2) DRAW (ADDR [LOGICAL8], ADDR [LOGICAL8], INTEGER, LOGICAL64)

This procedure is used when a batch of diagrams are to be drawn on a non-interactive output device. Selected charts from the input file P1 are drawn on to either a lineprinter or a graphical output device as specified by document name P2. A list of titles for the required charts should appear on the currently selected input stream, separated by newlines and terminated by the character ‘@’ at the start of a line. The word ALL will suffice if it is

required to draw all flowcharts from the specified file.

P1 – name of the input file. If this is left unspecified, the current file is used. If no current file is defined, this procedure will expect the Flowchart encodings to be on the current stream following the list of titles.

P2 – name of the output document. Zero means the current file.

P3 – the required level number, as with FLIP. In addition -1 may be specified in which case each diagram will be drawn at all existing levels.

P4 – specifies the device type on which the flowcharts are to be drawn as follows

- = 0 ordinary printer (LPT)
- = BEN Benson Plotter
- = GEN Genisco Display
- = HPP Hewlett Packard Plotter
- = MOT Motorola Display
- = KER Device on another machine

This determines the format of the output in the document P2.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 12

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

12 GRAPHICAL FACILITIES

The graphical facilities for application software in MUSS are primarily based on GKS. A full implementation of GKS is the objective. A list of procedure specifications for GKS together with information on language binding is given in this Chapter. For a full description of GKS the user is referred to ISO DIS 7942 “The Kernel System (GKS)” or one of the several textbooks available on the subject. Further graphical facilities for software such as the CORE graphics system are described in Volume 3.

Some MUSS workstations are equipped with raster scan graphics devices. For example, all the VME/10 workstations have a raster scan terminal and some have laser printers which are memory mapped. Therefore this Chapter also describes some ‘driver level’ facilities to support the scan conversion of characters and a limited amount of the graphics requirements. These facilities are not normally used directly by the user, but by the device drivers of GKS, text formatting packages and some screen editors.

12.1 GKS LANGUAGE BINDING

The binding used in the implementation built into the MUSS system has been chosen to satisfy the following requirements

- 1) It should follow as closely as possible the GKS report (ISO DIS 7942 “The Graphics Kernel System (GKS)”).
- 2) It should be directly usable, albeit with minor difficulties from any of the languages supported by MUSS, and from the command languages.
- 3) It should be easily possible to create a MUSS library to conform to any other GKS binding as a straightforward set of procedures calling the built-in procedures.

A representation of the data types of ISO DIS 7942 is used as follows

GKS TYPE	MUSS TYPE
I Integer	INTEGER
R Real	REAL
S String	ADDR [LOGICAL8]
P Point	TYPE P is REAL X, Y
N Name	LOGICAL
E Enumerator Type	LOGICAL
n*R REALS	ADDR [REAL]
n*P POINTS	ADDR [P]

In order to avoid problems with languages which do not support parameter passing by pointers with variable bounds, a string will be either the smaller of the size of the bound or the number of characters up to the first 0 byte. In the case of other bounded pointers, e.g. n*P, the GKS procedures in which they appear have an integer parameter giving the value of n, therefore the bound is not used in the implementation. Some other procedures have fixed sized arrays, e.g. 4*R. These are also passed as bounded pointers but the bound is again not used. Two dimensional arrays are passed as pointers to the corresponding one dimensional arrays, in which it is assumed that the elements are stored row by row. The values of enumeration types are allocated systematically from 0 in the order given in ISO DIS 7942.

There are two exceptions to the rule that names map into LOGICAL, two of which occur in OPEN.WORKSTATION. These parameters define

the connection identifier
and the workstation type.

In this implementation the connection identifier is an ADDR [LOGICAL8] which is only relevant for workstations connected as named I/O devices, e.g. various penplotters. The parameter must specify the MUSS stream name assigned to the device e.g.

```
% "BEN*"
% "HPP*"
```

This is the MUSL terminology for string literals, in other languages the format will be different. When the workstation is also the logged in terminal a dummy stream should be specified, i.e., % "*". This applies in the case

of VME/10 monitors and Tektronics terminals. The error file parameter of OPEN.GKS could have been treated similarly, but it is in fact ignored and errors are always reported in the current stream.

The following procedure specifications indicate the procedure interface into the GKS implementation at the time of Issue 12. This may change as internationally agreed language bindings emerge.

12.1.1 Control Function Specifications

A description of the actions of this group of functions is contained in Section 52 Control Functions of GKS Version 7.2 dated 14th November 1982 (ISO DIS 7942).

```
OPEN.GKS (LOGICAL, INTEGER)
CLOSE.GKS ()
ERROR (INTEGER)
OPEN.WORKSTATION (LOGICAL, ADDR [LOGICAL8], ADDR [LOGICAL8])
CLOSE.WORKSTATION (LOGICAL)
ACTIVATE.WORKSTATION (LOGICAL)
DEACTIVATE.WORKSTATION (LOGICAL)
CLEAR.WORKSTATION (LOGICAL, LOGICAL)
REDRAW.ALL.SEGMENTS.ON.WORKSTATION (LOGICAL)
UPDATE.WORKSTATION (LOGICAL, LOGICAL)
SET.DEFERRAL.STATE (LOGICAL, LOGICAL, LOGICAL)
MESSAGE (LOGICAL, ADDR [LOGICAL8])
ESCAPE (LOGICAL, ADDR [LOGICAL8])
```

12.1.2 Segment Attributes

A description of the actions of this group of functions is contained in Section 5.6.2 Segment Attributes in GKS Version 7.2 dated 14th November 1982 (ISO DIS 7942).

```
SET.SEGMENT.TRANSFORMATION (LOGICAL, ADDR [REAL])
SET.VISIBILITY (LOGICAL, LOGICAL)
SET.HIGHLIGHTING (LOGICAL, LOGICAL)
SET.SEGMENT.PRIORITY (LOGICAL, REAL)
SET.DETECTABILITY (LOGICAL, LOGICAL)
```

12.1.3 Segment Manipulation Functions

A description of the actions of this group of functions is contained in Section 5.6.1 Segment Manipulation Functions of GKS Version 7.2 dated 14th November 1982 (ISO DIS 7942).

```
GKS.CREATE.SEGMENT (LOGICAL)
CLOSE.SEGMENT ()
RENAME.SEGMENT (LOGICAL, LOGICAL)
DELETE.SEGMENT (LOGICAL)
DELETE.SEGMENT.FROM.WORKSTATION (LOGICAL, LOGICAL)
ASSOCIATE.SEGMENT.WITH.WORKSTATION (LOGICAL, LOGICAL)
COPY.SEGMENT.TO.WORKSTATION (LOGICAL, LOGICAL)
INSERT.SEGMENT (LOGICAL, ADDR [REAL])
```

12.1.4 Output Functions

A description of the actions of this group of functions is contained in Section 5.3 Output Functions of GKS Version 7.2 dated 14th November 1982 (ISO DIS 7942).

```
POLYLINE (INTEGER, ADDR [P])
POLYMARKER (INTEGER, ADDR [P])
TEXT (P, P, ADDR [LOGICAL8])
FILL.AREA (INTEGER, ADDR [P])
CELL.ARRAY (P, P, P, P, INTEGER, INTEGER, ADDR [INTEGER])
```

12.1.5 Initialisation of Input Devices

A description of the actions of this group of functions is contained in Section 5.7.1 Initialisation of Input Devices of GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942).

```
INITIALISE.LOCATOR (LOGICAL, INTEGER, P, INTEGER, INTEGER,
  ADDR [REAL], ADDR)
INITIALISE.STROKE (LOGICAL, INTEGER, INTEGER, ADDR [P], INTEGER,
  INTEGER, ADDR [REAL], ADDR)
INITIALISE.VALUATOR (LOGICAL, INTEGER, REAL, INTEGER, ADDR [REAL],
  ADDR)
INITIALISE.CHOICE (LOGICAL, INTEGER, INTEGER, INTEGER,
  ADDR [REAL], ADDR)
```



```
INITIALISE.PICK (LOGICAL, INTEGER, LOGICAL, LOGICAL, LOGICAL,  
                INTEGER, ADDR [REAL], ADDR)  
INITIALISE.STRING (LOGICAL, INTEGER, LOGICAL, INTEGER, ADDR [REAL],  
                  ADDR)
```

12.1.6 Setting Mode of Input Devices

A description of the actions of this group of functions is contained in Section 5.7.2 Setting Mode of Input Devices in GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942).

```
SET.LOCATOR.MODE (LOGICAL, INTEGER, LOGICAL, LOGICAL)  
SET.STROKE.MODE (LOGICAL, INTEGER, LOGICAL, LOGICAL)  
SET.VALUATOR.MODE (LOGICAL, INTEGER, LOGICAL, LOGICAL)  
SET.CHOICE.MODE (LOGICAL, INTEGER, LOGICAL, LOGICAL)  
SET.PICK.MODE (LOGICAL, INTEGER, LOGICAL, LOGICAL)  
SET.STRING.MODE (LOGICAL, INTEGER, LOGICAL, LOGICAL)
```

12.1.7 Request Input Function

A description of the actions of this group of functions is contained in Section 5.7.3 Request Input functions in GKS Version 7.2 dated 14th November 1982 (ISO DIS 7942).

```
REQUEST.LOCATOR (LOGICAL, INTEGER, ADDR LOGICAL, ADDR INTEGER,  
                ADDR P)  
REQUEST.STROKE (LOGICAL, INTEGER, ADDR LOGICAL, ADDR INTEGER,  
                ADDR INTEGER, ADDR [P])  
REQUEST.VALUATOR (LOGICAL, INTEGER, ADDR LOGICAL, ADDR REAL)  
REQUEST.CHOICE (LOGICAL, INTEGER, ADDR LOGICAL, ADDR INTEGER)  
REQUEST.PICK (LOGICAL, INTEGER, ADDR LOGICAL, ADDR LOGICAL,  
              ADDR LOGICAL)  
REQUEST.STRING (LOGICAL, INTEGER, ADDR LOGICAL, ADDR [LOGICAL8])
```

12.1.8 Sample Input Functions

A description of the actions of this group of functions is contained in Section 5.7.4 Sample Input Functions of GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942).

```
SAMPLE.LOCATOR (LOGICAL, INTEGER, ADDR INTEGER, ADDR P)
```

SAMPLE.STROKE (LOGICAL, INTEGER, ADDR INTEGER, ADDR INTEGER,
ADDR [P])
SAMPLE.VALUATOR (LOGICAL, INTEGER, ADDR REAL)
SAMPLE.CHOICE (LOGICAL, INTEGER, ADDR INTEGER)
SAMPLE.PICK (ADDR[LOGICAL8], INTEGER, ADDR LOGICAL, ADDR LOGICAL,
ADDR LOGICAL)
SAMPLE.STRING (LOGICAL, INTEGER, ADDR [LOGICAL8])

12.1.9 Event Input Functions

A description of the actions of this group of functions is contained in Section 5.7.5 Event Input Functions of GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942) .

AWAIT.EVENT (REAL, LOGICAL, ADDR LOGICAL, ADDR INTEGER)
FLUSH.DEVICE.EVENTS (LOGICAL, LOGICAL, INTEGER)
GET.LOCATOR (ADDR INTEGER, ADDR P)
GET.STROKE (ADDR INTEGER, ADDR INTEGER, ADDR [P])
GET.VALUATOR (ADDR REAL)
GET.CHOICE (ADDR INTEGER)
GET.PICK (ADDR LOGICAL, ADDR LOGICAL, ADDR LOGICAL)
GET.STRING (ADDR [LOGICAL8])

12.1.10 Workstation Independent Primitive Attributes

A description of the actions of this group of functions is contained in Section 5.4.1 Workstation Independent Primitive Attributes in GKS Version 7.2 dated 14th November 1982 (ISO DIS 7942).

SET.POLYLINE.INDEX (INTEGER)
SET.LINE.TYPE (INTEGER)
SET.LINEWIDTH.SCALE.FACTOR (REAL)
SET.POLYLINE.COLOUR.INDEX (INTEGER)
SET.POLYMARKER.INDEX (INTEGER)
SET.MARKER.TYPE (INTEGER)
SET.MARKERSIZE.SCALE.FACTOR (REAL)
SET.POLYMARKER.COLOUR.INDEX (INTEGER)
SET.TEXT.INDEX (INTEGER)
SET.TEXT.FONT.AND.PRECISION (INTEGER, LOGICAL)
SET.CHARACTER.EXPANSION.FACTOR (REAL)
SET.CHARACTER.SPACING (REAL)

SET.TEXT.COLOUR.INDEX (INTEGER)
SET.CHARACTER.HEIGHT (REAL)
SET.CHARACTER.UP.VECTOR (ADDR [REAL])
SET.TEXT.PATH (LOGICAL)
SET.TEXT.ALIGNMENT (LOGICAL, LOGICAL)
SET.FILL.AREA.INDEX (INTEGER)
SET.FILL.AREA.INTERIOR.STYLE (LOGICAL)
SET.FILL.AREA.STYLE.INDEX (INTEGER)
SET.FILL.AREA.COLOUR.INDEX (INTEGER)
SET.PATTERN.SIZE (ADDR [REAL])
SET.PATTERN.REFERENCE.POINT (P)
SET.ASPECT.SOURCE.FLAGS (ADDR [LOGICAL])
SET.PICK.IDENTIFIER (LOGICAL)
SET.COLOUR.REPRESENTATION (LOGICAL, INTEGER, ADDR [REAL])

12.1.11 Normalisation Transformation

A description of the actions of this group of functions is contained in Section 5.5.1 Normalisation Transformation in GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942).

SET.WINDOW (INTEGER, ADDR [REAL])
SET.VIEWPORT (INTEGER, ADDR [REAL])
SET.VIEWPORT.INPUT.PRIORITY (INTEGER, INTEGER, LOGICAL)
SELECT.NORMALIZATION.TRANSFORMATION (INTEGER)
SET.CLIPPING.INDICATOR (LOGICAL)

12.1.12 Workstation Transformation

A description of the actions of this group of functions is contained in Section 5.5.2 Workstation Transformation in GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942).

SET.WORKSTATION.WINDOW (LOGICAL, ADDR [REAL])
SET.WORKSTATION.VIEWPORT (LOGICAL, ADDR [REAL])

12.1.13 Utility Functions

A description of the actions of this group of functions is contained in Section 5.10 Utility Functions in GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942)

```
EVALUATE.TRANSFORMATION.MATRIX (P, ADDR [REAL], REAL, ADDR [REAL],
    LOGICAL, ADDR [REAL])
ACCUMULATE.TRANSFORMATION.MATRIX (ADDR[REAL], P, ADDR [REAL],
    REAL, ADDR [REAL], LOGICAL, ADDR [REAL])
```

12.1.14 Inquiry Functions for Operating State Value

A description of the actions of this group of functions is contained in Section 5.9.2 Inquiry Functions for Operating State Value in GKS Version 7.2 dated 14th November 1982 (ISO DIS 7942)

```
INQUIRE.OPERATING.STATE.VALUE (ADDR LOGICAL)
```

12.1.15 Inquiry Functions for GKS Description Table

A description of the actions of this group of functions is contained in Section 5.9.5 of Inquiry Functions for GKS Description Table in GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942).

```
INQUIRE.LEVEL.OF.GKS (ADDR INTEGER, ADDR LOGICAL)
INQUIRE.LIST.OF.AVAILABLE.WORKSTATION.TYPES (ADDR INTEGER,
    ADDR INTEGER, ADDR [LOGICAL])
INQUIRE.WORKSTATION.MAXIMUM.NUMBERS (ADDR INTEGER, ADDR INTEGER,
    ADDR INTEGER, ADDR INTEGER)
INQUIRE.MAXIMUM.NORMALIZATION TRANSFORMATION.NUMBER
    (ADDR INTEGER, ADDR INTEGER)
```

12.1.16 Inquiry Functions for GKS State List

A description of the actions of this group of functions is contained in Section 5.9.4 Inquiry Functions for GKS State List of GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942).

```
INQUIRE.SET.OF.OPEN WORKSTATIONS (ADDR INTEGER, ADDR INTEGER,
    ADDR [LOGICAL])
INQUIRE.SET.OF.ACTIVE.WORKSTATIONS (ADDR INTEGER, ADDR INTEGER,
    ADDR [LOGICAL])
INQUIRE.CURRENT.PRIMITIVE.ATTRIBUTE.VALUES (ADDR INTEGER,
    ADDR INTEGER, ADDR INTEGER, ADDR INTEGER, ADDR REAL,
    ADDR [REAL], ADDR LOGICAL, ADDR [LOGICAL], ADDR INTEGER,
    ADDR [REAL], ADDR P, ADDR LOGICAL)
```

```

INQUIRE.CURRENT.INDIVIDUAL.ATTRIBUTE VALUES (ADDR INTEGER,
  ADDR INTEGER, ADDR REAL, ADDR INTEGER, ADDR INTEGER,
  ADDR REAL, ADDR INTEGER, ADDR INTEGER, ADDR LOGICAL,
  ADDR REAL, ADDR REAL, ADDR INTEGER, ADDR LOGICAL,
  ADDR INTEGER, ADDR INTEGER, ADDR [LOGICAL])
INQUIRE.CURRENT.NORMALIZATION.TRANSFORMATION.NUMBER
  (ADDR INTEGER, ADDR INTEGER)
INQUIRE.LIST.OF NORMALIZATION.TRANSFORMATION.NUMBERS
  (ADDR INTEGER, ADDR [INTEGER])
INQUIRE.NORMALIZATION.TRANSFORMATION (INTEGER, ADDR INTEGER,
  ADDR [REAL], ADDR [REAL])
INQUIRE.CLIPPING.INDICATOR (ADDR INTEGER, ADDR LOGICAL)
INQUIRE.NAME.OF.OPEN.SEGMENT (ADDR INTEGER, ADDR LOGICAL)
INQUIRE.SET.OF.SEGMENT.NAMES.IN.USE (ADDR INTEGER, ADDR INTEGER,
  ADDR [LOGICAL])
INQUIRE.MORE.SIMULTANEOUS.EVENTS (ADDR INTEGER, ADDR LOGICAL)

```

12.1.17 Inquiry Functions for Workstations

A description of the actions of this group of functions is contained in Section 5.9.6 Inquiry Functions for Work:station Description Table of GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942)

```

INQUIRE.NO.OF.SEGMENT.PRIORITIES.SUPPORTED (ADDR [LOGICAL8],
  ADDR INTEGER, ADDR INTEGER)
INQUIRE.DYNAMIC.MODIFICATION.OF.SEGMENT.ATTRIBUTES (ADDR [LOGICAL8],
  ADDR INTEGER, ADDR LOGICAL, ADDR LOGICAL, ADDR LOGICAL,
  ADDR LOGICAL, ADDR LOGICAL, ADDR LOGICAL, ADDR LOGICAL)
INQUIRE.NUMBER.OF.AVAILABLE.LOGICAL.INPUT DEVICES (ADDR [LOGICAL8],
  ADDR INTEGER, ADDR INTEGER, ADDR INTEGER, ADDR INTEGER,
  ADDR INTEGER, ADDR INTEGER, ADDR INTEGER)
INQUIRE.DEFAULT.LOCATOR.DEVICE.DATA (ADDR [LOGICAL8], INTEGER,
  ADDR INTEGER, ADDR P, ADDR [INTEGER], ADDR [INTEGER],
  ADDR [REAL], ADDR ADDR)
INQUIRE.DEFAULT.STROKE.DEVICE.DATA (ADDR [LOGICAL8], INTEGER,
  ADDR INTEGER, ADDR INTEGER, ADDR [INTEGER], ADDR [INTEGER],
  ADDR [REAL], ADDR ADDR)
INQUIRE.DEFAULT.VALUATOR.DEVICE.DATA (ADDR [LOGICAL8], INTEGER,
  ADDR INTEGER, ADDR REAL, ADDR [INTEGER], ADDR [INTEGER],
  ADDR [REAL], ADDR ADDR)

```

```

INQUIRE.DEFAULT.CHOICE.DEVICE.DATA (ADDR [LOGICAL8], INTEGER,
    ADDR INTEGER, ADDR INTEGER, ADDR [INTEGER], ADDR [INTEGER],
    ADDR [REAL], ADDR ADDR)
INQUIRE.DEFAULT.PICK.DEVICE.DATA (ADDR [LOGICAL8], INTEGER,
    ADDR INTEGER, ADDR [INTEGER], ADDR [INTEGER], ADDR [REAL],
    ADDR ADDR)
INQUIRE.DEFAULT.STRING.DEVICE.DATA (ADDR [LOGICAL8], INTEGER,
    ADDR INTEGER, ADDR INTEGER, ADDR [INTEGER], ADDR [INTEGER],
    ADDR [REAL], ADDR ADDR)
INQUIRE.SET.OF.ASSOCIATED.WORKSTATIONS (LOGICAL, ADDR INTEGER,
    ADDR INTEGER, ADDR [LOGICAL])
INQUIRE.SEGMENT.ATTRIBUTES (LOGICAL, ADDR INTEGER, ADDR [REAL],
    ADDR LOGICAL, ADDR LOGICAL, ADDR REAL, ADDR LOGICAL)

```

12.1.18 Pixel Inquiries

A description of the actions of this group of functions is contained in Section 5.9.8 Pixel Inquiries in GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942).

```

INQUIRE.PIXEL.ARRAY.DIMENSION (LOGICAL, P, P, ADDR INTEGER,
    ADDR [INTEGER])
INQUIRE.PIXEL.ARRAY (LOGICAL, P, ADDR (INTEGER), ADDR INTEGER,
    ADDR LOGICAL, ADDR [INTEGER])
INQUIRE.PIXEL (LOGICAL, P, ADDR INTEGER, ADDR INTEGER)
INQUIRE.INPUT.QUEUE.OVERFLOW (ADDR INTEGER, ADDR LOGICAL,
    ADDR LOGICAL, ADDR INTEGER)

```

12.1.19 Inquiry Functions for Workstation State List

A description of the actions of this group of functions is contained in Section 5.9.5 Inquiry Functions for Workstation State List in GKS Version 7.2 dated 14th November 1982 (ISO DIS 7942).

```

INQUIRE.WORKSTATION.CONNECTION.AND.TYPE (LOGICAL, ADDR INTEGER,
    ADDR [LOGICAL8], ADDR [LOGICAL8])
INQUIRE.WORKSTATION.STATE (LOGICAL, ADDR INTEGER, ADDR LOGICAL)
INQUIRE.WORKSTATION.DEFERRAL.AND.UPDATE.STATES (LOGICAL,
    ADDR INTEGER, ADDR LOGICAL, ADDR LOGICAL, ADDR LOGICAL,
    ADDR LOGICAL)
INQUIRE.LIST.OF.POLYMARKER.INDICES (LOGICAL, ADDR INTEGER,

```

```
    ADDR INTEGER, ADDR [INTEGER])
INQUIRE.POLYLINE.REPRESENTATION (LOGICAL, INTEGER, LOGICAL,
    ADDR INTEGER, ADDR INTEGER, ADDR REAL, ADDR INTEGER)
INQUIRE.LIST.OF.POLYMARKER.INDICES (LOGICAL, ADDR INTEGER,
    ADDR INTEGER, ADDR [INTEGER])
INQUIRE.POLYMARKER.REPRESENTATION (LOGICAL, INTEGER, LOGICAL,
    ADDR INTEGER, ADDR INTEGER, ADDR REAL, ADDR INTEGER)
INQUIRE.LIST.OF.TEXT.INDICES (LOGICAL, ADDR INTEGER, ADDR INTEGER,
    ADDR [INTEGER])
INQUIRE.TEXT.REPRESENTATION (LOGICAL, INTEGER, LOGICAL,
    ADDR INTEGER, ADDR INTEGER, ADDR LOGICAL, ADDR REAL,
    ADDR REAL, ADDR INTEGER)
INQUIRE.TEXT.EXTENT (LOGICAL, P, ADDR [LOGICALS], ADDR INTEGER,
    ADDR P, ADDR [P])
INQUIRE.LIST.OF.FILL.AREA.INDICES (LOGICAL, ADDR INTEGER,
    ADDR INTEGER, ADDR [INTEGER])
INQUIRE.FILL.AREA.REPRESENTATION (LOGICAL, INTEGER, LOGICAL,
    ADDR INTEGER, ADDR LOGICAL, ADDR INTEGER, ADDR INTEGER)
INQUIRE.LIST.OF.PATTERN.INDICES (LOGICAL, ADDR INTEGER, ADDR INTEGER,
    ADDR [INTEGER])
INQUIRE.PATTERN.REPRESENTATION (LOGICAL, INTEGER, LOGICAL,
    ADDR INTEGER, ADDR [INTEGER], ADDR [INTEGER])
INQUIRE.LIST.OF.COLOUR.INDICES (LOGICAL, ADDR INTEGER, ADDR INTEGER,
    ADDR [INTEGER])
INQUIRE.COLOUR.REPRESENTATION (LOGICAL, INTEGER, LOGICAL,
    ADDR INTEGER, ADDR [REAL])
INQUIRE.WORKSTATION.TRANSFORMATION (LOGICAL, ADDR INTEGER,
    ADDR LOGICAL, ADDR [REAL], ADDR [REAL], ADDR [REAL],
    ADDR [REAL])
INQUIRE.SET.OF.SEGMENT.NAMES.ON.WORKSTATION (LOGICAL, ADDR INTEGER,
    ADDR INTEGER, ADDR [LOGICAL])
INQUIRE.LOCATOR.DEVICE.STATE (LOGICAL, INTEGER, LOGICAL,
    ADDR INTEGER, ADDR LOGICAL, ADDR LOGICAL, ADDR INTEGER,
    ADDR P, ADDR INTEGER, ADDR [REAL], ADDR ADDR)
INQUIRE.STROKE.DEVICE.STATE (LOGICAL, INTEGER, LOGICAL,
    ADDR INTEGER, ADDR LOGICAL, ADDR LOGICAL, ADDR INTEGER,
    ADDR INTEGER, ADDR [P], ADDR INTEGER, ADDR [REAL], ADDR ADDR)
INQUIRE.VALUATOR.DEVICE.STATE (LOGICAL, INTEGER, ADDR INTEGER,
    ADDR LOGICAL, ADDR LOGICAL, ADDR REAL, ADDR INTEGER,
    ADDR [REAL], ADDR ADDR)
```

```

INQUIRE.CHOICE.DEVICE.STATE (LOGICAL, ADDR LOGICAL, ADDR LOGICAL,
    ADDR LOGICAL, ADDR LOGICAL, ADDR INTEGER, ADDR INTEGER,
    ADDR [REAL], ADDR ADDR)
INQUIRE.PICK.DEVICE.STATE (LOGICAL, INTEGER, LOGICAL, ADDR INTEGER,
    ADDR LOGICAL, ADDR LOGICAL, ADDR LOGICAL, ADDR LOGICAL,
    ADDR LOGICAL, ADDR INTEGER, ADDR [REAL], ADDR ADDR)
INQUIRE.STRING.DEVICE.STATE (LOGICAL, INTEGER, ADDR INTEGER,
    ADDR LOGICAL, ADDR LOGICAL, ADDR [LOGICAL8], ADDR INTEGER,
    ADDR [REAL], ADDR ADDR)

```

12.1.20 Inquiry Functions for Workstations Description Table

A description of the actions of this group of functions is contained in Section 5.9.6 Inquiry Functions for Workstation Description Table of GKS Version 7.2 dated 14th. November 1982 (ISO DIS 7942).

```

INQUIRE.WORKSTATION.CATEGORY (ADDR [LOGICAL8], ADDR INTEGER,
    ADDR LOGICAL)
INQUIRE.WORKSTATION.CLASSIFICATION (ADDR [LOGICAL8], ADDR INTEGER,
    ADDR LOGICAL)
INQUIRE.MAXIMUM.DISPLAY.SURFACESIZE (ADDR [LOGICAL8], ADDR INTEGER,
    ADDR LOGICAL, ADDR [REAL], ADDR [INTEGER])
INQUIRE.DYNAMIC.MODIFICATION.OF.WORKSTATION.ATTRIBUTES
    (ADDR [LOGICAL8], ADDR INTEGER, ADDR LOGICAL, ADDR LOGICAL,
    ADDR LOGICAL, ADDR LOGICAL, ADDR LOGICAL, ADDR LOGICAL,
    ADDR LOGICAL)
INQUIRE.DEFAULT.DEFERRAL.STATE.VALUES (ADDR [LOGICAL8],
    ADDR INTEGER ADDR LOGICAL, ADDR LOGICAL)
INQUIRE.POLYLINE.FACILITIES (ADDR [LOGICAL8], ADDR INTEGER,
    ADDR INTEGER, ADDR [INTEGER], ADDR INTEGER, ADDR REAL,
    ADDR [REAL], ADDR INTEGER)
INQUIRE.PREDEFINED.POLYLINE REPRESENTATION (ADDR [LOGICAL8],
    INTEGER, ADDR INTEGER, ADDR INTEGER, ADDR REAL, ADDR INTEGER)
INQUIRE.POLYMARKER.FACILITIES (ADDR [LOGICAL8], ADDR INTEGER,
    ADDR INTEGER, ADDR [INTEGER], ADDR INTEGER, ADDR REAL,
    ADDR [REAL], ADDR INTEGER)
INQUIRE.PREDEFINED.POLYMARKER.REPRESENTATION (ADDR [LOGICAL8],
    INTEGER, ADDR INTEGER, ADDR INTEGER, ADDR REAL, ADDR INTEGER)
INQUIRE.TEXT.FACILITIES (ADDR [LOGICAL8], ADDR INTEGER, ADDR INTEGER,
    ADDR [INTEGER], ADDR [LOGICAL], ADDR INTEGER, ADDR REAL,

```



```

      ADDR REAL, ADDR INTEGER, ADDR REAL, ADDR REAL, ADDR INTEGER)
INQUIRE.PREDEFINED.TEXT.REPRESENTATION (ADDR [LOGICAL8], INTEGER,
      ADDR INTEGER, ADDR INTEGER, ADDR LOGICAL, ADDR REAL,
      ADDR REAL, ADDR INTEGER)
INQUIRE.FILL.AREA.FACILITIES (ADDR [LOGICAL8], ADDR INTEGER,
      ADDR INTEGER, ADDR [LOGICAL], ADDR INTEGER, ADDR [INTEGER],
      ADDR INTEGER)
INQUIRE.PREDEFINED.FILL.AREA.REPRESENTATION (ADDR [LOGICAL8],
      ADDR INTEGER, ADDR INTEGER, ADDR LOGICAL, ADDR INTEGER,
      ADDR INTEGER)
INQUIRE.PATTERN.FACILITIES (ADDR [LOGICAL8], ADDR INTEGER,
      ADDR INTEGER)
INQUIRE.PREDEFINED.PATTERN.REPRESENTATION (ADDR [LOGICAL8],
      ADDR INTEGER, ADDR INTEGER, ADDR [INTEGER], ADDR [INTEGER])
INQUIRE.COLOUR.FACILITIES (ADDR [LOGICAL8], ADDR INTEGER,
      ADDR INTEGER, ADDR LOGICAL, ADDR INTEGER)
INQUIRE.PREDEFINED.COLOUR.REPRESENTATION (ADDR [LOGICAL8], INTEGER,
      ADDR INTEGER, ADDR [REAL])
INQUIRE.LIST.OF.AVAILABLE.GENERALISED.PRIMITIVES (ADDR [LOGICAL8],
      ADDR INTEGER, ADDR INTEGER, ADDR [LOGICAL])
INQUIRE.GENERALISED.DRAWING.PRIMITIVE (ADDR [LOGICAL8],
      ADDR INTEGER, ADDR INTEGER, ADDR [LOGICAL])
INQUIRE.MAXIMUM.LENGTH.OF.WORKSTATION.STATE.TABLE (ADDR [LOGICAL8],
      ADDR INTEGER, ADDR INTEGER, ADDR INTEGER, ADDR INTEGER,
      ADDR INTEGER, ADDR INTEGER, ADDR INTEGER)

```

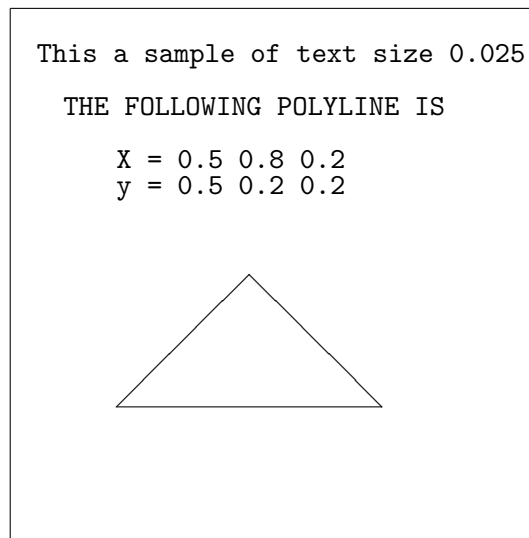
12.2 USING GKS FROM FORTRAN

The mapping of the Fortran parameter types into MUSS types is as follows, this correspondence of types is mainly straightforward except for the user defined type P which corresponds to a pair of reals. This ploy is possible because the Fortran compiler treats Complex as a type containing two reals.

MUSS TYPE	FORTRAN TYPE
INTEGER	INTEGER
REAL	REAL
ADDR [LOGICAL8]	CHARACTER
TYPE P IS REAL X, Y	COMPLEX

LOGICAL	INTEGER
LOGICAL	INTEGER
ADDR [REAL]	REAL
ADDR [P]	COMPLEX

An example will perhaps be useful. The following diagram is the screen image generated by the Fortran program given below



```

*ANSISWITCH
*IMPORT 'OPENGKS', 'OPENWORKSTATION', 'ACTIVATEWORKSTATION'
*IMPORT 'DEACTIVATEWORKSTATION', 'CLOSEWORKSTATJON ', 'CLOSEGKS'
*IMPORT 'POLYLINE', 'TEXT', 'RL', 'LIB'
*IMPORT 'SETCHARACTERHEIGHT', 'SETPOLYLINECOLOURINDEX'
COMPLEX PNTS (4)
PNTS (1) = (0.5, 0.5)
PNTS (2) = (0.8, 0.2)
PNTS (3) = (0.2, 0.2)
PNTS (4) = (0.5, 0.5)
CALL RL ('UTILGKS', 0)
CALL LIB ('UTILGKS', 0)
CALL OPENGKS (0, 0)
CALL OPENWORKSTATION ( 1, '*', 'VME')
CALL ACTIVATEWORKSTATION (1)

```

```
CALL SETPOLYLINECOLOURINDEX (4)
CALL POLYLINE (4, PNTS)
CALL SETCHARACTERHEIGHT (0.025)
CALL TEXT ((0.05,0.92), 'This is a sample of text size 0.025')
CALL TEXT ((0.1,0.84), 'THE FOLLOWING POLYLNE IS')
CALL TEXT ((0.17,0.75), 'X = 0.5 0.8 0.2')
CALL TEXT ((0.17,0.71), 'Y = 0.5 0.2 0.2')
CALL DEACTIVATEWORKSTATION (1)
PAUSE 'TYPE ''CONTINUE'' TO CLOSE WORKSTATION'
CALL CLOSEWORKSTATION (1)
CALL CLOSEGKS ()
END
```

12.3 USING GKS FROM PASCAL

The full Pascal binding to GKS is available in the form of a library of procedures written in Pascal. These procedures conform to the ISO/TC97/SC5/WG2 Version 5.1 dated 7th. August 1984, “Information Processing Systems Computer Graphics Graphical Kernel System (GKS) Language Bindings”, Part 2: Pascal. However, as an alternative, the procedures described in [Section 12.1](#) can also be called directly from a Pascal program.

The mapping of Pascal parameter types into MUSS types is as below, and the correspondence of types is mainly straightforward. A point must be declared as a user defined type, e.g.

```
type grpoint = record
    x, y : real
end;
```

Arrays correspond to arrays with either conformant or fixed bounds. It is generally more convenient to use conformant arrays.

MUSS TYPE	PASCAL TYPE
INTEGER	Integer
REAL	Real
ADDR [LOGICAL8]str	: packed array [min .. max : integer] of char
P	grpoint
LOGICAL	scalar type
ADDR [REAL]	var xx : array [min .. max : integer] of real
ADDR [D]	var pp : array [min .. max : integer] of grpoint

The following points should be noted:

1. The GKS MUSS library must be open at compile time and run time. A single command LIB UTIL:GKS achieves this.
2. Many GKS procedures with simple parameters can be called without giving an explicit external declaration. However, procedures with ADDR [REAL], ADDR [P] and ADDR [LOGICAL8] parameters must be given a full external declaration. The ADDR [REAL] and ADDR [P] parameters should be var parameters but ADDR [LOGICAL8] parameters are more conveniently treated as conformant value parameters as this permits literal strings to be used as actual parameters.
3. Warnings given by the compiler on such external lines can usually be ignored but errors must not be ignored.
4. The smallest literal string that can be substituted for ADDR [LOGICAL8] is two characters, e.g. '*', as a single character string literal is of the wrong type in Pascal and must therefore be written as '* '. The smallest array that can be handed over will contain one character.
5. The TEXT procedure should be called by the name TX as otherwise the name is confused with the Pascal TEXT data type.
6. The dots in GKS procedure names must be omitted. Upper or lower case letters can be used for procedure names as Pascal treats both as upper case.
7. 32-bit real arithmetic must be selected. This is the default in Pascal.

A Pascal program to reproduce the diagram in Section 12.2 is:

```

program gksexample (input, output);
type
    grpoint = record
        x, y : real
    end;
var
    pnts : array [1..4] of grpoint;
    position : grpoint;
    procedure polyline (n : integer;
        var pp : array [min..max : integer] of grpoint);external;
    procedure tx (position : grpoint;
        str : packed array [min..max : integer] of char);external;
begin
    pnts [1] . x := 0.5;      pnts [1] . y := 0.5;
    pnts [2] . x := 0.8;      pnts [2] . y := 0.2;
    pnts [3] . x := 0.2;      pnts [3] . y := 0.2;
    pnts [4] . x := 0.5;      pnts [4] . y := 0.5;
    rl ('UTIL:GKS',0);
    lib ('UTIL:GKS',0);
    opengks (0,0);
    open workstation (1, '*', 'VME');
    activateworkstation (1);
    setpolylinecolourindex (4);
    polyline (4, pnts);
    setcharacterheight (0 025);
    position .x := 0.05; position .y = 0.92;
    tx (position, 'This is a sample of text size 0.025');
    position .x := 0.1; position .y := 0.84;
    tx (position, 'The following polyline is');
    position .x := 0.17; position .y := 0.75;
    tx (position, 'x := 0.5 0.8 0.2');
    position .x := 0.17; position .y := 0.71;
    tx (position, 'y = 0.5 0.2 0.2');
    deactivateworkstation (1);
    waitr (1);
    closeworkstation (1);
    closegks;
end

```

12.4 GKS WORKSTATIONS

The GKS Implementation is structured so that it is easy to vary the number of workstations and to add new kinds of workstations, and it is expected that most MUSS systems will differ in the workstations which they support. Any system which attempts to support graphics seriously will provide either a penplotter or laser printer for hard copy and an 'OUTIN' workstation for interactive use.

Several GKS workstations may be mapped onto an actual workstation providing they do not have input devices. Each may have its own viewport to produce the effect of 'windows' So that this multiple window facility may be used on graphics terminals, they have two entries in the list below.

WORKSTATION TYPES ON THE VME/10

VME	console as output only
VME+	console with input
TEK	terminal as output only
TEK+	terminal with input
BEN	Benson plotter
HPP	Hewlett Packard plotter
LAS	laser printer

12.5 GKS ERROR MESSAGES

The GKS specification Version 7.2 dated 14th. November 1982 (ISO DIS 7942), Annex B defines error numbers for the fault condition which an implementation should detect. These numbers are reproduced below

12.5.1 States

- 1 GKS not in proper state: GKS shall be in the state GKCL
- 2 GKS not in proper state: GKS shall be in the state GKOP
- 3 GKS not in proper state: GKS shall be in the state WSAC
- 4 GKS not in proper state: GKS shall be in the state SGOP
- 5 GKS not in proper state: GKS shall be either in the state WSAC or in the state SGOP
- 6 GKS not in proper state: GKS shall be either in the state WSOP or in the state WSAC

- 7 GKS not in proper state: GKS shall be in one of the states
 WSOP, WSAC or SGOP
- 8 GKS not in proper state: GKS shall be in one of the states
 GKOP, WSOP, WSAC or SCOP

12.5.2 Workstations

- 20 Specified workstation identifier is invalid
- 21 Specified connection identifier is invalid
- 22 Specified workstation type is invalid
- 23 Specified workstation type does not exist
- 24 Specified workstation is open
- 25 Specified workstation is not open
- 26 Specified workstation cannot be opened
- 27 Workstation independent Segment Storage is not open
- 28 Workstation Independent Segment Storage is already open
- 29 Specified workstation is active
- 30 Specified workstation is not active
- 31 Specified workstation is of category MO
- 32 Specified workstation is not of category MO
- 33 Specified workstation is of category MI
- 34 Specified workstation is not of category MI
- 35 Specified workstation is of category INPUT
- 36 Specified workstation is Workstation Independent Segment
 Storage
- 37 Specified workstation is not of category OUTIN
- 38 Specified workstation is neither of category INPUT nor of
 category OUTIN
- 39 Specified workstation is neither of category OUTPUT nor of
 category OUTIN
- 40 Specified workstation has no pixel store read back capability
- 41 Specified workstation type is not able to generate the
 specified generalised drawing primitive

12.5.3 Transformations

- 50 Transformation number is invalid
- 51 Rectangle definition is invalid
- 52 Viewport is not within the Normalized Device Coordinate unit
 square

- 53 Workstation window is not within the Normalized Device
 Coordinate unit square
- 54 Workstation viewport is not within the display space

12.5.4 Output Attributes

- 60 Polyline index is invalid
- 61 A representation for the specified polyline index has not
 been defined on this workstation
- 62 Linetype is less than or equal to zero
- 63 Specified linetype is not supported on this workstation
- 64 Polymarker index is invalid
- 65 A representation for the specified polymarker index has not
 been defined on this workstation
- 66 Marker type is less than or equal to zero
- 67 Specified marker type is not supported on this workstation
- 68 Text index is invalid
- 69 A representation for the specified text index has not been
 defined on this workstation
- 70 Text font is less than or equal to zero
- 71 Requested text font is not supported for the the specified
 precision on this workstation
- 72 Character expansion factor is less than or equal to zero
- 73 Character height is less than or equal to zero
- 74 Length of character up vector is zero
- 75 Fill area index is invalid
- 76 A representation for the specified full area index has not
 been defined on this workstation
- 77 Specified fill area interior style is not supported on this
 workstation
- 78 Style (pattern or hatch) index is less than or equal to zero
- 79 Specified pattern index is invalid
- 80 Specified hatch style is not supported on this workstation
- 81 Pattern size value is not positive
- 82 A representation for the specified pattern index has not
 been defined on this workstation
- 83 Interior style PATTERN is not supported on this workstation
- 84 Dimensions of colour array are invalid
- 85 Colour index is less than zero
- 86 Colour index is invalid

- 87 A representation for the specified colour index has not been
 defined on this workstation
- 88 Colour is outside range (0,1)
- 89 Pick identifier is invalid

12.5.5 Output Primitives

- 100 Number of points is invalid
- 101 Invalid code in string
- 102 Generalised drawing primitive identifier is invalid
- 103 Content of generalised drawing primitive data record is
 invalid
- 104 At least one active workstation is not able to generate the
 specified generalised drawing primitive

12.5.6 Segments

- 120 Specified segment name is invalid
- 121 Specified segment name is already in use
- 122 Specified segment does not exist
- 123 Specified segment does not exist on specified workstation
- 124 Specified segment does not exist on Workstation Independent
 segment Storage
- 125 Specified segment is open
- 126 Segment priority is outside the range(0,1)

12.5.7 Input

- 140 Specified input device is not present on workstation
- 141 Input device is not in REQUEST mode
- 142 Input device is not in SAMPLE mode
- 143 EVENT and SAMPLE input mode are not available at this
 level of GKS
- 144 Specified prompt and echo type is not supported on this
 workstation
- 145 Echo area is outside display space
- 146 Contents of input data record are invalid
- 147 Input queue has overflowed
- 148 Input queue has not overflowed since GKS was opened or
 the last invocation of INQUIRE INPUT QUEUE OVERFLOW
- 149 Input queue has overflowed but associated workstation has

- been closed
- 150 No input value of the correct class is in the current event report
- 151 Timeout is invalid
- 152 Initial value is invalid
- 153 Length of Initial string is greater than the implementation defined maximum

12.5.8 Metafiles

- 160 Item type is not allowed for user items
- 161 Item length is invalid
- 162 No item is left in GKS metafile input
- 183 Metafile item is invalid
- 164 Item type is not a valid GKS item
- 165 Content of item data record is invalid for the specified item type
- 166 Maximum item data record length is invalid
- 167 User item cannot be interpreted

12.5.9 Escape

- 180 Specified function is not supported
- 181 Contents of escape data record are invalid

12.5.10 Implementation Dependent

- 300 Specified function is not supported in this level of GKS
- 301 Storage overflow has occurred in GKS
- 302 Storage overflow has occurred in segment storage
- 303 Input/Output error has occurred while reading
- 304 Input/Output error has occurred while writing
- 305 Input/Output error has occurred while sending data to a workstation
- 306 Input/Output error has occurred while receiving data from a workstation
- 307 Input/Output error has occurred during program library management
- 308 Input/Output error has occurred while reading workstation description table
- 309 Arithmetic error has occurred

12.6 RASTER SCAN DEVICE FACILITIES

These facilities cater for line drawing and character generation in various font styles on raster scan devices. The position of lines and text are specified in pixels within the pixel array of raster scan devices and the drawing procedures maintain a current drawing position. The coordinate system for this array is such that pixel (0, 0) corresponds to the bottom left of the device view surface

1) SELECT.GRA (ADDR [LOGICAL8])

This procedure selects the raster scan device to which subsequent output operations of this section apply until another device is selected. P1 specifies the device type as follows

VME — VME/10 graphic terminal (800 x 600 pixels)
LAS — VME Laser printer (2048 x 3136 pixels).

2) LINE (INTEGER, INTEGER, INTEGER, INTEGER)

This procedure moves the current position to the position specified by (P2, P3) and performs the drawing action specified by P4. P1 specifies whether the new position (P2, P3) is an absolute position (P1 = 0) or relative to the current position (P1 = 1). If only one coordinate of an absolute position needs changing then the other ordinates may be set to -32767. P4 specifies drawing action:

-1 — draw the cursor at (P2, P3)
0 — no drawing action
1 — draw a solid line
2 — draw a dashed line
3 — draw a dotted line
4 — draw a dash dotted line.

3) DEFINE.FONT (INTEGER, INTEGER, REAL, REAL, REAL, REAL, REAL)

There are a number of predefined fonts available and they are numbered as follows, together with examples

```

0 - '!'#$%&'()=?~|1234567890ABCDEFGHIJKL
MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz{
xyz{@[*]+;:;],.-><

1 - '!'£$%&'()=?~|1234567890ABCDEFGHIJKL
MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz{
xyz{@[*]+;:;],.-><

2 - '!'£$%&'()=?~|1234567890ABCDEFGHIJKL
MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz{
xyz{@[*]+;:;],.-><

3 - '!'£$%&'()=?~|1234567890ABCDEFGHIJKL
MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz{
xyz{@[*]+;:;],.-><

4 - '!'£$%&'()=?~|1234567890ABCDEFGHIJKL
MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz{
xyz{@[*]+;:;],.-><

```

Fonts 5 to 15 are reserved for future predefined fonts.

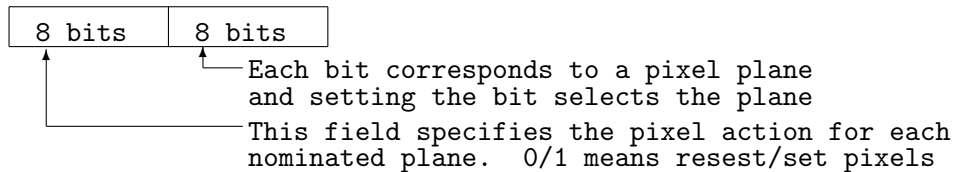
Users may define their own fonts by calling NEW.FONT. When a predefined or user defined font is selected for text output the characters are of a predefined size and the direction of text is horizontal from left to right. To generate text with a different size and direction DEFINE.FONT is first called to define a new font style. The parameters specify the mapping from a defined font to the required font style. P1 is the predefined font number. P2 the new font style number (16 to 63), P3 the intercharacter gap in pixels and P4, P5, P6, P7 the coefficients of the matrix that specifies the scaling and direction of the vectors which form each character in the font (CXX, CYY, CXY and CYX).

4) GRA.CMD (INTEGER, INTEGER)

The action of this procedure depends on P1 as follows:

- 0 — select font style P2 for textual output
- 1 — clears pixel store or raster scan device
- 2 — for raster scan hard copy devices, this option initiates
 the hard copy output of the pixel store

- 3 – this option switches text bolding on and off: the first call turns holding on, and the second call turns holding off
- 4 – this option switches the underlining of text on and off
- 5 – select superscripting for the current font, and save (on a stack) the previous script position
- 6 – select subscripting for the current font, and save the previous script position
- 7 – select the script position from the stack of saved script positions
- 8 – save the current drawing position on a stack
- 9 – set the current drawing position from the stack of saved drawing positions
- 10 – for raster scan devices with multiple pixel planes, this option selects the planes that are operated upon by subsequent drawing commands. For each nominated plane the option also specifies whether pixels are set or reset. P2 is encoded as follows



- 11 – this resets all options to their default values
- 12,13 – this specifies the alignment of text with respect to the current drawing position. Option 12 selects the x ordinate

- 1 - normal
- 2 - left
- 3 - centre
- 4 - right

and option 13 selects the y ordinate alignment

- 1 - normal
- 2 - top

- 3 - capital
 - 4 - half
 - 5 - base
 - 6 - bottom
- 14 — option selects the text path direction (P2)
- 1 - right
 - 2 - left
 - 3 - up
 - 4 - down
- 15 — this option increases the current drawing position by half a linefeed
- 16 — this option reduces the current drawing position by half a linefeed
- 17 — deletes that part of the text line to right of the current drawing position
- 18 — this option enables the cursor
- 19 — this option disables the cursor
- 20 — this option outputs the character P2 at the current drawing position. The result gives the width (pixels) of the character
- 21 — this option inserts the character P2 into a line of text at the point specified by the current drawing position. The result gives the width (pixels) of the inserted character
- 22 — this option deletes the character P2 from a line of text whose position is given by the current drawing position. The result gives the width (pixels) of the deleted character
- 23 — this option moves the drawing position over the character P2 that is at the current drawing position

- 24 – this option moves the drawing position back over the character P2 that ends at the current drawing position
- 25 – this option enables scrolling
- 26 – this option disables scrolling
- 27 – this option moves the drawing position to the start of the next line, and scrolling may occur if it is operative. The result gives the height (pixels) of the next line
- 28 – this option deletes the line of text whose position is given by the current drawing position
- 29 – this option resets the defaults for clipping
- 30 – P2 specifies the number of copies to print
- 31 – this option selects the current marker character (P2)
- 32 – this option specifies the size (P2) in pixels of the marker
- 33 – this option specifies the thickness (P2) in pixels of the lines.

5) GRA.TEXT (ADDR [LOGICAL8])

This procedure outputs text on the device in the currently selected font style. The drawing position is not updated. The text P1 may contain embedded commands as described in [12.6.1](#)

6) GRA.TEXT.EXTENT (ADDR [LOGICAL8], INTEGER, INTEGER,
 ADDR [REAL], ADDR [REAL]) / INTEGER

This procedure determines the extent in pixels of the text string P1 relative to the first character when drawn in font style P2 and updates the X and Y coordinate vectors, P4 and P5, with the following points of a notional rectangle that encloses the text:

Element 0 – lower start point
Element 1 – lower start point for concatenated text
Element 2 – upper start point
Element 3 – upper end point
Element 4 – lower end point.

The text path direction is given by P3 as follows:

1 – right
2 – left
3 – up
4 – down.

The result gives the length in pixels of the text string. If the points of the enclosing text rectangle are not required then P4 and P5 can be nil pointers.

7) IN.GRA.TEXT ()

This procedure outputs on the raster scan device text from the currently selected input stream. As with GRA.TEXT, the input may contain embedded commands as described in [12.6.1](#). The principal use of this procedure is by processes dedicated to managing raster scan devices, such as the LASER process of [12.6.1](#).

8) FONT.ENQ (INTEGER USER TYPE, USER TYPE, USER TYPE)

The procedure returns pointers to three vectors that describe how the characters (starting with the character whose code is %20) or the predefined font style P1 are drawn.

The type of the elements in these vectors is as follows:

LOGICAL8 (CODE vector)
INTEGER16 (INDEX vector)
LOGICAL8 (WIDTH vector).

Pointers to these vectors are returned via parameters P2, P3 and P4 respectively. The WIDTH vector contains the width in pixels of characters. The CODE vector contains a list of line segment points for drawing each character, while the INDEX vectors give the start of these lists within CODE.

The character line segments are for a rectangle of pixels whose bottom left coordinate is (0, 0).

The lists in CODE contain a sequence of pixel coordinates and a line segment is drawn between consecutive pairs of points unless the most significant bit of the X ordinate of the second point is set. The point (255, 255) indicates the end of the line segment list.

9) NEW.FONT (INTEGER, ADDR [LOGICAL8], ADDR [INTEGER16])

This procedure specifies a user defined font whose number is P1. P2 and P3 specify how to draw the characters in the font; these parameters are similar to P2 and P3 of FONT.ENQ, respectively.

10) OUTPUT.FONT (INTEGER)

This outputs on the currently selected output stream the user defined font P1 as an escape sequence command (see [12.6.1](#)).

11) GRA.CLIP (INTEGER, INTEGER, INTEGER, INTEGER, INTEGER)

This procedure specifies the clip rectangle for the device. (P1, P2) specifies the coordinate for the lower left corner point and (P3, P4) the upper right corner point of the clip rectangle. The viewport height is given by P5.

12) MARKERS (ADDR [INTEGER16])

This procedure draws markers at absolute positions. The parameter (P1) is a vector whose elements are in pairs that specify the coordinate position of the required markers.

12.6.1 The Laser Process

A command LASER exists which forms the body of a process which may be established in any MUSS system to which the laser printer hardware is attached. It acts as a device driver and will print all documents sent to it on its channel 0. Thus for example, if the process is called LASER flowcharts may be drawn by the command

```
DRAW file LASER* 1 LAS
```

All characters except escape sequences are copied straight to the printer and the escape sequences control, for example, graphics output and font selection.

In addition to the standard fonts available escape sequences are provided to make new fonts. These new fonts are normally derived from old fonts with different character sizes and orientations specified. Completely new fonts may be specified in vector form. Fonts are numbered upwards from 0 and 16 for standard and document defined fonts, respectively.

The facilities provided by escape sequences are:

Define a font with different text parameters	(ESC A)
Select a font	(ESC B)
Define a new font	(ESC I)
Move cursor to an absolute position	(ESC C)
Move cursor relative to current position	(ESC D)
Draw a line to an absolute position	(ESC E)
Draw a line to a relative position	(ESC F)
Stack current position	(ESC G)
Reset cursor position from stack	(ESC H)
Stack script position and select superscript	(ESC J)
Stack script position and select subscripting	(ESC K)
Reset script position from stack	(ESC L)
Switch text bolding on and off	(ESC M)
Switch text underlining on and off	(ESC N)
Select pixel planes	(ESC O)
Reset device options	(ESC P)
Set text path direction	(ESC Q)
Set text alignment	(ESC R)
Up half a line feed	(ESC S)
Down half a line feed	(ESC T)
Set the clip rectangle	(ESC U)
Output text	(ESC V)
Reset to clip default	(ESC W)
Set number of copies	(ESC X)
Set marker character	(ESC Y)
Set marker size	(ESC Z)
Draw markers	(ESC a)
Set line thickness	(ESC b)

The actions of most of the above should be self-evident. The following notes relate to the more complicated ones.

1) Defining a new font (ESC A)

This sequence defines a new font based on an existing font style. It has seven parameters. Parameter P1 is the number of new font. P2 is the font number of character style. P3 is the gap between characters specified in pixels and P4 to P7 specify the transformation between the pixel matrix for each character of the font and their actual pixel representation. The four coefficients are CXX, CYY, CXY and CYX.

2) Drawing a line to an absolute position (ESC E)

This command draws a line from the current cursor position to (P1, P2), and then sets the cursor position to (P1, P2), P3 specifies the linetype:

- 1 – Solid
- 2 – Dashed
- 3 – Dotted
- 4 – Dash dotted.

3) Drawing a line to a relative position (ESC F)

This command draws a line to the point which has a position of (P1, P2) relative to the current cursor position, and updates the cursor position. P3 specifies the linetype.

4) Setting the text alignment (ESC R)

This command specifies the text alignment; P1 and P2 give the text alignment in the x and y direction respectively. See options 12, 13 of GRA.CMD.

Parameters for the escape sequences are given as a sequence of bytes. For multi-byte parameters the bytes are in an ascending order of significance. In the list below, the parameters for each escape sequence are specified as MUSL types, thus an INTEGER16 parameter would be specified by 2 bytes. For vector pointer parameters, a two byte element count precedes the list of elements.

<ESC A> (LOGICAL8, LOGICAL8, REAL, REAL32, REAL32, REAL32,
REAL32)
<ESC B> (LOGICAL8)
<ESC I> (LOGICAL8, VECTOR OF LOGICAL8, VECTOR OF INTEGER16)
<ESC C> (INTEGER16, INTEGER16)
<ESC D> (INTEGER16, INTEGER16)
<ESC E> (INTEGER16, INTEGER16, LOGICAL8)
<ESC F> (INTEGER16, INTEGER16, LOGICAL8)
<ESC G> ()
<ESC H> ()
<ESC J> ()
<ESC K> ()
<ESC L> ()
<ESC M> ()
<ESC N> ()
<ESC O> (INTEGER16)
<ESC P>
<ESC Q> (LOGICAL8)
<ESC S>
<ESC T>
<ESC U> (INTEGER16, INTEGER16, INTEGER18, INTEGER16, INTEGER16),
(INTEGER16)
<ESC V> (INTEGER16, VECTOR OF LOGICAL8)
<ESC W> ()
<ESC X> (INTEGER16)
<ESC Y> (LOGICAL8)
<ESC Z> (INTEGER16)
<ESC a> (INTEGER16, VECTOR OF INTEGER16)
<ESC b> (INTEGER16)

MUSS

USER MANUAL

VOLUME 1

CHAPTER 13

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

13 VIRTUAL STORE MANAGEMENT

Each process in MUSS is provided with its own virtual store. The virtual store is assumed to be segmented, and a set of procedures are provided to enable a process to create/release segments and change their attributes. The exact number and limiting sizes of the segments varies between MUSS installations, and not all of the segments may be directly accessible. For this reason, two types of segment are distinguished.

The first type is for areas which must be randomly accessible, such as for code or stack/buffer areas. In general, these will be demand loaded by the operating system. and may be accessed by generating an appropriate machine address. On some machines, particularly small (16 bit) machines, the actual addressing capability of the hardware is too restrictive. A process is therefore allowed a larger number of segments than can be addressed directly, and facilities are provided to allow a user to MAP a subset of the segments into the addressable space. (The exact number is machine dependent.) The MAP function is also used on machines which are unable to support demand loading, to inform the system which segments must be in store whilst the process is running.

The second type of segment is suitable for uses in which there is a well defined access pattern, such as for input/output documents. This type of segment is known as an X.SEGMENT, and is characterised mainly by the fact that its maximum size is not determined by the physical restrictions of the central processor. For example, on the PDP11, an X.segment may be created which is very much larger than the addressable space of a process (64 Kbytes). These segments are notionally disc based, and as they cannot be addressed directly, the only way in which they may be accessed is by copying individual blocks into/out of accessible segments. A command, COPY.BLOCK, is provided for this purpose. In all other respects, the two types of segment are treated in a similar way, and the commands described in this chapter to enable a process to manipulate its virtual store, will operate on either type of segment. Also either type of segment may be sent as a message or filed.

A process is initially created with only one segment (segment 0). A process may not change any of the properties of this segment or release it as it is generally used as a stack and to hold the global variables, such as

the PW's. A process is free to use other segments for the stack, and the organisational commands will operate using the current stack area.

Further segments may be created by the user process, and subsequently the process can change the size or access permission of the segments or release them to recover the space. There are six access permission bits associated with each segment, denoted TAURWO.

- T – Task copy bit. When set, the segment is copied on creating a new task. Files opened with this access permission are also copied at the time the file is opened.
- A – Authorised to change. When this is set (= 1) the process is allowed to increase its access permission to the segment.
- U – User permission bit. When set (= 1) the segment is accessible by the process when in user mode, otherwise only the operating system may access it.
- R – Read permission bit. If set, the process may read the segment as data.
- W – Write permission bit. If set, the process may alter the contents of the segment.
- O – Obey permission bit. If set, the process may execute instructions from the segment.

When a segment is initially created, it is cleared to zero and has an access permission of %1E i.e. read and write access in user mode. with authorisation to change the permission.

13.1 COMMANDS

1) CREATE.SEGMENT (INTEGER, ADDR)

This procedure creates a new segment on behalf of the current process. Parameter P1 specifies the segment number. If this is negative, then the system will choose a suitable segment number to allocate. The second parameter P2 is the segment size in bytes. This might be rounded up by the system to some suitable multiple of the page size. The maximum size

of a segment is, of course, restricted by the address translation hardware. For example. on MU6G this size is 256K bytes. If the user requests a size greater than this, consecutive segments will be allocated to cover the required area. It should be noted, however, that in all subsequent operations, multiple segments are treated as separate entities and not as one contiguous area. If P2 is negative or zero, a single segment of maximum size will be created.

The create segment procedure returns two values. The segment number is returned in PW1. (This value is returned both when the system selects a segment number or when the number is specified in P1.) If multiple consecutive segments are allocated, PW1 contains the number of the lowest segment assigned. PW2 returns the size of the area allocated in bytes. This may differ from P2 if rounding has taken place.

2) CREATE.X.SEGMENT (INTEGER)

This procedure creates an X.segment on behalf of the current process. Parameter P1 is the segment size in pages. (The page size is installation dependent.)

The procedure returns two values. A segment identifier is returned in PW1, and this should be used in all subsequent commands when referring to the segment. The size allocated for the segment (in pages) is returned in PW2. As for create segment, this may differ from P1 if rounding has taken place.

3) RELEASE.SEGMENT (INTEGER)

This procedure removes the specified segment P1 from the users virtual store. If a file has been opened into this segment, the action of releasing it is in effect to close the file for the current process.

4) CHANGE.SIZE (INTEGER, ADDR)

This procedure changes the size of the segment specified in P1 to the size given by P2. If the segment is an X.segment, P2 is the new size in pages, otherwise it is the size in bytes. As with the create segment command the size may again be rounded by the system. If the size is ≤ 0 , the effect is to release the segment. It should be noted that changing the size upwards to

greater than the maximum segment size is invalid, and does not result in the creation of subsequent segments to allow for the upward expansion.

The size of the segment after this command is returned in PW1.

5) CHANGE.ACCESS (INTEGER, LOGICAL)

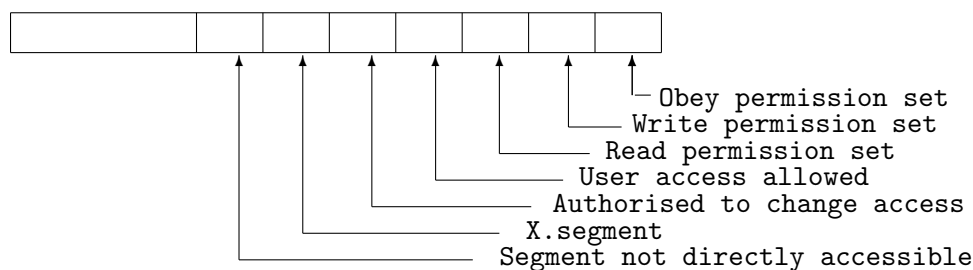
This procedure changes the access permission of the segment specified in P1. The access permission P2 is bit significant and is interpreted as the TAURWO bits explained earlier. If the A bit is currently set for the segment, the process is allowed to change its access to P2. If the A bit is not currently set, then only reductions in access permission are allowed, and parameter P2 is verified accordingly.

6) INTERCHANGE (INTEGER, INTEGER)

This procedure interchanges the specification of the two segments given by the parameters. It is permissible for either (or even both) of the segments to be undefined, and in this case the effect is simply to redefine the existing segment. Segments may not be interchanged with X.segments.

7) READ.SEGMENT.STATUS (INTEGER)

This procedure reads the status of the segment specified, and returns in PW1 and PW2 the size in bytes (pages for an X segment) and access permission respectively. The access permission is in the form:



Note, this command does not supply the task copy bit.

8) MAP (INTEGER, INTEGER, INTEGER)

This procedure is used primarily on machines with small or otherwise unusual virtual stores to ensure that a segment is accessible. P3 specifies the number of segments to be made accessible, starting at segment number P1. P2 indicates how the segment may be mapped in the virtual store.

On machines with small virtual stores (e.g. PDP11), this procedure maps the segment into part of the addressable space. In this case, P2 specifies a segment position within the addressable space. It should also be used with P1 = 0, when a segment is to be removed completely from the addressable space.

On machines, such as the MC68010 or VAX, with a large virtual store which can be demand paged, this procedure has no effect other than setting the PW results.

The procedure sets PW1 to the number of the segment which previously occupied that area of the virtual store. PW6 is set to the segment number to be used in address calculations to access the segment. This procedure has no effect if an X.segment is specified in P1.

9) SECURE.SEGMENT (INTEGER)

This command causes the paging system to update the disc copies of all the altered pages in the segment specified by P1.

10) COPY.BLOCK (INTEGER, INTEGER, INTEGER, INTEGER)

This procedure copies a page of information between two segments, and in particular, it should be used to copy information between X.segments and mapped segments wherever the X.segments have to be accessed. P1 and P2 specify a segment number and page number which is the source of the data. P3 and P4 specify the destination segment and page number.

11) CHANGE.COMMON.SEGMENT (INTEGER, INTEGER)

This procedure takes the local segment specified by P1 and causes it to become the common segment specified by P2. It applies to all processes and it lasts until the system is restarted. There are a limited number of common segments which the command may specify and they are determined by the

local configuration state of each machine. If the common segment already exists, this segment will become the local segment P1.

12) PASS.SEGMENT (INTEGER, INTEGER, INTEGER, INTEGER)

This procedure enables a process to pass a segment from within its virtual store directly to another process. The second process must have been created by the current process and must be in a suspended state (see Chapter 14). P1 is the number of the segment to be passed and this segment is removed from the virtual store of the current process. P2 is the number of the segment in the destination process. P3 and P4 are the SPN and PID respectively of the destination process.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 14

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

14 CREATION AND CONTROL OF PROCESS

MUSS is designed as a system in which any process may create other processes and act as their supervisor. It is thus possible to extend the facilities of the basic system, by creating additional/alternative supervisor processes. In this Chapter, the facilities for creating and controlling processes are described.

14.1 PROCESS CREATION

Process creation is achieved by calling a command `CREATE.PROCESS`. Any process may use this command provided that it can supply the username and password of an authorised user of the system. The user will subsequently be charged for the resources used by the process. The creating process becomes the supervisor of the created process, and as such, it is allowed to use a special set of commands for controlling the process.

14.2 SCHEDULING

When initially created, a process is in a suspended state and is not eligible for scheduling by the operating system. The supervisor may activate the process using the `FREE.PROCESS` command, and the process is then eligible for CPU time. The allocation of CPU time is based on the priority level associated with the process at the time of its creation. Process priorities are non-negative integers with 0 being the highest priority and the largest integer the lowest. In general the highest priority process which is free to run is allocated CPU time.

Processes with identical priority are considered in order of creation. The only exception is priority 31, where the processes are cyclically allocated timeslices.

14.3 CPU TIME

The maximum amount of CPU time to be made available to a process is specified (in seconds) at the time a process is created. A process can arrange to be interrupted after a specified number of milliseconds has elapsed by calling `SET.TIMER`. This provides the means of subdividing the available time between various parts of a job. On being interrupted, the timer is

automatically set to the maximum amount of time remaining. Note that a small amount of the total available time is normally withheld by the system for the purpose of monitoring in the event of using the requested amount of CPU time. The total amount of CPU time used may also be read using the READ.TIMER command. The accuracy of the timer, particularly with respect to the microsecond timer, is subject to machine dependent tolerances.

14.4 COMMANDS

1) CREATE.PROCESS (LOGICAL64, LOGICAL64, LOGICAL64, INTEGER, INTEGER, ADDR START.PROC, INTEGER, INTEGER)

This procedure creates a process of name P1 on behalf of the user whose name and password are given in P2 and P3 respectively. The process will run at the priority given by P7 and will commence execution at the procedure P6.

The maximum resources to be given to the process include a store limit P4 specified in K words (integers), and a CPU limit of P5 seconds.

The facility is also provided for the system to inform the supervisor when a process terminates. The parameter P8 defines a channel number on which the supervisor is prepared to accept a termination message containing the process identification. This channel number is encoded as for send message.

The identification of a process takes the form of a system process number (SPN) and a process identifier (PID). These are returned in PW1 and PW2 by the create process command, and must subsequently be used by the supervisor when calling the procedures for controlling the process.

2) TERMINATE.PROCESS (INTEGER)

This procedure is called whenever a process wishes to terminate. The parameter states a termination reason, which will be sent to the supervisor if a termination message was requested at create time. Negative fault reasons imply termination due to a fault condition. In particular, a reason of -100 implies that the process was aborted, and the system will attempt to salvage whatever monitoring information is available.

3) KILL (LOGICAL64, INTEGER)

This command may be called by a user to terminate the specified process P1. It will only operate if the process is running under the same username as the current user (or if the current user is privileged). The parameter P2 states a reason for killing the process.

4) SUSPEND.PROCESS (INTEGER, INTEGER)

This procedure may be called by a process' supervisor to suspend execution of the specified process (P1 = SPN, P2 = PID).

5) FREE.PROCESS (INTEGER, INTEGER)

This command allows a process' supervisor to free it, and thus make it eligible for scheduling (P1 = SPN, P2 = PID).

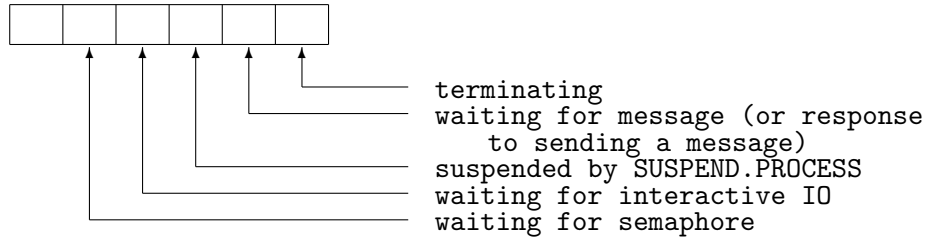
6) LOOK.UP.PROCESS (LOGICAL64, LOGICAL64)

This procedure finds the internal identification of the process specified in P1. If P1 is zero, the identification of the current process is returned. The second parameter states the machine in which the process exists (0 implying the current machine). The identification takes the form of an SPN in PW1 and a PID in PW2.

7) READ.PROCESS.STATUS (INTEGER)

This procedure reads the status of the process with the specified SPN. If this is negative, the status of the current process is returned. The status information takes the form of

PWW1	Process name
PWW2	Username
PW1	SPN
PW2	Process status, which is non zero if the process is suspended.
PW3	Priority
PW4	PID
PW5	Suspension reason. which is bit significant with the following Interpretation



PW6 Start time.

8) CATALOGUE.PROCESSES ()

This procedure returns a segment containing information about all the processes in the machine. The entries, which may be indexed by SPN, occupy 32 bytes each and have the following format:

Process name (8 bytes)
User name (8 bytes)
1 spare byte
Suspension reason (1 byte)
Status (1 byte)
Priority (1 byte)
CPU time used
Process identifier PID
Unused

Each entry is 4 bytes unless stated otherwise. The total number of entries is returned in PW1, and PW2 holds the segment number.

9) SET.TIMER (INTEGER32)

This procedure allows a process to set its local CPU timer. The parameter states the number of milliseconds before the process will be interrupted (trap 2). To allow for cases where this may be greater than the total amount of CPU time remaining, the timer is set equal to the smaller of the two.

10) READ.TIMER ()

This command returns in PW1 the amount of CPU time in seconds used

by the current process. The maximum available time in seconds is also returned in PW2. The time remaining in milliseconds in the local CPU time is returned in PW3. The number of milliseconds used by the current process is returned in PW4.

11) FORCE.INT (INTEGER, INTEGER, LOGICAL)

This command may be called to force an external interrupt (interrupt trap 3) on a process. P1 and P2 specify the SPN and PID of the process respectively, and P3 a reason to be passed to the trap procedure. The command will only operate if the current process is the supervisor of the process being interrupted or if the current user is privileged.

14.5 MULTI-TASKING WITHIN A PROCESS

A facility exists for supporting multiple threads of execution within a single MUSS process. In this instance, each thread is referred to as a *task* of the process, and each process consists of at least one such task. Whenever a new task is created, its environment is identical to the currently executing task. Segments will be copied or shared according to the access permission specified for each segment, and the stack segment will always be replicated. Thereafter, a task is able to change its environment (for example, create segments, open files etc.) without directly affecting the other tasks. N.B. The onus is on the user to achieve correct synchronisation between the various tasks when manipulating shared data structures.

Each task may be identified by its *Task Number*, and a process is created running task number zero. Whenever a new task is created, its number is returned to enable it subsequently to be scheduled. Although the creation of tasks may notionally follow a hierarchical organisation, the system regards each task as an independent thread of execution and does not maintain information regarding the parentage of each task. Scheduling of the individual tasks is entirely the responsibility of the user process as the basic mechanisms only provide for re-entering a specified task, not for selecting the number of the task to be entered.

The following procedures exist for managing multiple tasks

1) CREATE.TASK ()

This procedure creates a new task environment identical to the current task but with its own stack segment. PW1 may be used to distinguish between the calling task (i.e. the parent) and the newly created task (the child). In the case of the parent, PW1 returns the task number of the new task; in the case of the child, PW1 is zero. Other segments in the virtual store will be either shared or copied when the new task is created according to the access permission associated with each segment (see Chapter 13).

2) DELETE.TASK (INTEGER)

This procedure removes the task, P1, from the current process, and recovers all resources used solely by that task. If P1 specifies the current task, task zero is reselected.

3) TASK (INTEGER)

This procedure resumes the task specified by P1. It is anticipated that this will be called by some higher level synchronisation/scheduling activity within the process.

4) CURRENT.TASK ()

This procedure returns in PW1 the number of the current task.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 15

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

15 INTER-PROCESS COMMUNICATIONS

Processes in MUSS normally communicate with one another by sending messages and the input/output devices connected to the system also generate and receive messages. Each process has eight input channels to which messages may be directed. The identification of the destination for a message is therefore a combination of the identification of the process and a channel number for that process. Also included is information concerning the machine in which the process resides, thus allowing messages to be passed across a network. Specially encoded process identifiers are used to address peripheral devices, but in all other respects, message passing between processes and between a process and a peripheral is identical.

To enable a two-way communication to be set up easily, and to prevent a rogue process from injecting unidentified messages into the communication system, the system appends to each message the identification of the sender. This is returned to the destination process when it reads the message, and may subsequently be used as the address for reply messages. There is an additional problem in ensuring that messages sent as replies can be associated with a particular message from the original process. Processes can therefore specify a sequence number, which is presented to the destination process as part of the reply information.

A process also has control over which messages are accepted on a channel. Each of the eight channels can be set into one of three states: closed, open for all messages, and dedicated to one specific process. Any message directed to a closed channel is rejected, any message sent to an open channel is accepted and automatically queued. Having eight channels under its control, therefore, a process can selectively stream its input.

Each of the message channels acts as a first-in-first-out queue. There are a number of ways in which a process can tell whether there are messages waiting to be read. A process may poll its message channels by attempting to read messages from them. Secondly, it may elect to wait until a message arrives on a specified channel, or one of a number of channels. In this case, a time limit is also specified, and the process is re-activated if no messages have arrived within that period. A process may choose to be interrupted when the next message arrives on a channel (Trap 3). Finally, a process may request the status and condition of one of its message channels.

A message consists of two parts, a short header optionally accompanied by a segment from the sender's virtual store. The header is copied from the sending to the receiving process' virtual store; the segment is unlinked from the sender's virtual store when the message is sent, and subsequently linked into the receivers virtual store when the message is read. The format of data in a message is not fixed by the system, and is usually by arrangement between the sending and receiving processes. The system software and input/output controllers have certain conventions for messages, and these are described in Section 15.3.

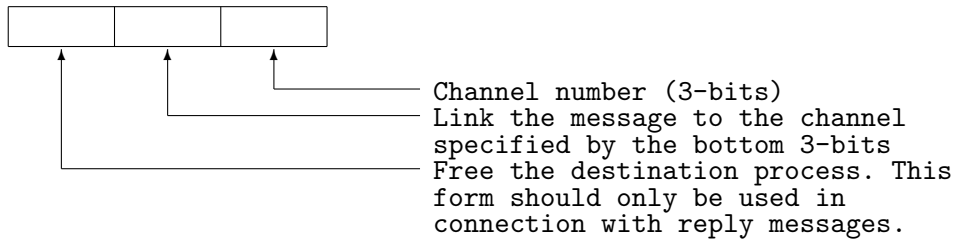
The identification of a process or peripheral as far as the communication system is concerned takes the form of a vector of four integers. This vector holds, respectively

the system process number (SPN) of the process
its process identifier (now redundant)
mode information and
a sequence number.

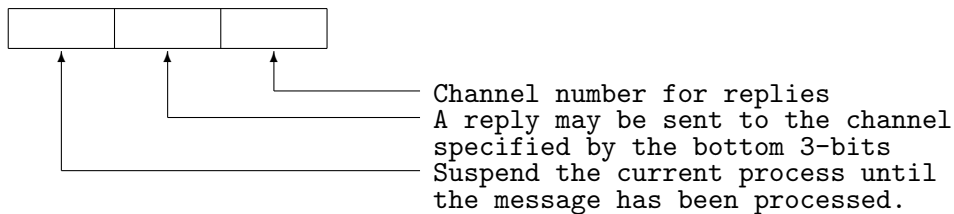
A vector of this type is used to specify the destination when sending a message or when connecting a peripheral to a process, and is returned to a process when reading a message to identify the source of the message.

The format of the SPN is described in Section 8.2.1 and Section 14.4. The process identifier (PID), appearing as element 2 in the vector, is logically redundant but its position is retained for compatibility with earlier systems. The mode information has two functions. In the case of the destination vector when sending a message, the mode identifies the channel on which the message is to be queued. A similar mode parameter is also specified to identify the synchronisation required by the sending process, namely whether a reply message is required or whether the sending process is to be suspended until the message has been processed. This mode is passed to the recipient process when reading the message and forms part of the source information. The encoding of this information is designed so that the source vector acquired at the time of reading a message may be used directly either to send a reply message or to free the suspended process in accordance with the requested form of synchronisation. The encoding of this mode is as follows:

a) When part of the destination vector



b) When used as the source mode



An alternative to the message system is provided for those situations where buffered input/output is unsatisfactory, for example, in a word processing package. Direct single character communication with a device is provided as described in Section [15.2](#)

15.1 MESSAGE COMMANDS

1) SEND.MESSAGE (ADDR [LOGICAL8], ADDR [INTEGER], INTEGER, INTEGER, INTEGER, LOGICAL)

This procedure sends a message to a specified channel of a process. P1 is the data to be sent in the short message. It should not be greater than 114 characters long, and should exist in store normally accessible to the calling process. If longer messages are included, they are truncated after 114 characters.

The second parameter, P2, is a vector of four elements specifying the destination for the message. P3 specifies the source mode and identifies the form of synchronisation required, as described earlier.

The source sequence number, P4, provides a mechanism whereby a process

can append an identifier on to an outgoing message. This identifier may then be returned as a destination sequence number by the recipient process when it replies. so that the original process can associate outgoing and reply messages. The source sequence number may take any user defined value.

The parameters P5 and P6 represent a segment number and access permission respectively, for a segment within the process' virtual store. The segment must be present in the virtual store, and the same rules apply to changes in access permission as would apply to a process calling the change access command. If the segment parameter is zero, then no segment is sent.

2) READ.MESSAGE (ADDR [LOGICAL], ADDR [INTEGER], INTEGER, INTEGER)

This procedure reads a message from the channel specified by the bottom 3 bits of P3. P1 is a vector where the characters passed as the short message may be returned. It should exist in store normally accessible to the calling process. If the vector is not long enough to store the short message, the excess characters will be discarded and lost.

The parameter P2 should be a vector of at least 4 entries. Information about the source of the message is copied into this vector, in the form:

SPN of the source
Reserved
Mode for replies
Sequence number for replies.

This, of course corresponds exactly with the destination parameter of the send message command. Thus, conversations may be set up by using this information to send reply messages.

The fourth parameter specifies a segment number which may be used for long messages. If this is negative, a free segment number is allocated. PW1 always contains the segment number actually used (zero if a short message). PW2 contains the size of the segment in bytes and PW3 holds the access permission accorded.

3) SET.CH.STATUS (INTEGER, INTEGER, INTEGER, INTEGER)

This procedure sets the status of the input channel specified by P1. The status, given in P2, is a 4-bit quantity interpreted in the following way:

	d	c	b	a
--	---	---	---	---

- (a) Allow messages from any process.
- (b) Dedicate the channel to allow messages from the process whose identifier (SPN) is given in P3.
- (c) Close the channel after 1 message corresponding to the specified sequence number P4. Reject all other messages.
- (d) Interrupt the process when the next message arrives. This option is reset when the interrupt is forced, and so a subsequent call of SET.CH.STATUS is required to reassert this option.

A status of zero implies that the channel is closed to all incoming messages.

4) WAIT (LOGICAL, INTEGER)

This procedure halts the current process pending the arrival of a message. The first parameter defines the channels on which messages are expected. Thus, if the least significant bit is set, messages on channel 0 will wake the process; if the next bit, messages on channel 1, etc.

To avoid a process waiting an indefinite period of time for a message, a time limit in seconds, P2, may be specified. If a message on one of the required channels has not been received within this time, then the process is woken.

If the time limit is negative or zero, or a message is already on one of the required channels, then the procedure returns immediately without halting the process. PW1 always contains a bit pattern indicating the channels on which messages are present. Thus, the least significant bit is set if messages are on channel 0, etc.

If P1 is zero this command unconditionally waits for P2 seconds

5) READ.CH.STATUS (INTEGER)

This procedure reads the status of the specified message channel. PW1 returns the status bits, as specified in SET.CH.STATUS, PW2 the SPN if the channel is dedicated and PW3 the sequence number if it is dedicated for a specific message. The number of messages on the channel is returned in PW4.

15.2 DIRECT COMMUNICATION COMMANDS

These commands provide an alternative more direct way of communicating with peripherals which are logged into a message channel.

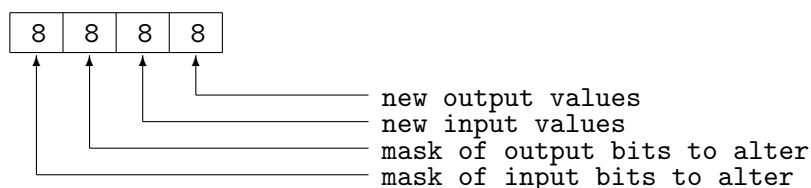
1) CHANGE.CHANNEL (INTEGER, INTEGER, LOGICAL32)

This procedure provides the ability to alter the behaviour of a peripheral.

P1 specifies the peripheral SPN of the channel. The SPN is available from the source information supplied when reading a message from a peripheral. It can also be obtained by calling I.SOURCE for any message stream which has received input from the peripheral.

P2 specifies the new communication mode of the device. The least significant bit selects direct IO (0) or message IO (1). In the direct IO case, the next bit determines the action when READ.CH is called and no character is available, namely whether the process is halted (0) or a value of -1 is returned (1). In the case of messages, it distinguishes whether short messages (0) or long messages (1) are required from the device. If the third bit is set, the channel will be non-dedicated, any process may output to it.

P3 specifies changes to the device protocol mode and identifies the mode bits to alter and the values to assign to those bits. The format of the parameter is:



If a bit in the mask field is set, a new value will be set for that mode bit. The significance of the bits are the same as in the CONFIG command described in Section 8.2.2.

PW1 contains the mode bits prior to the change channel command. The least significant eight bits contain the output mode and the next eight bits the input mode. The upper 16 bits are all 1's, thus allowing this value to be used in a subsequent call of CHANGE.CHANNEL to reset the device mode to its original state. PW2 contains the previous communication mode.

2) READ.CH (INTEGER)

This procedure reads the next character of input from the device specified by P1. P1 is a peripheral SPN, as used in CHANGE.CHANNEL for configuring the device for direct IO. The character is returned in the least significant a bits of PW1 and upper part of PW1 is a count of the number of characters already input, which remain to be read. If there is no input waiting in the buffer, and the P2 value used in CHANGE CHANNEL was 0, the called process will be suspended until the next input character arrives. If P2 in CHANGE.CHANNEL was 2 then -1 will be returned in PW1 whenever the buffer is empty.

15.3 FORMAT OF MESSAGES

In general, a process may send any information it wishes to another process, in a format mutually agreed by the sending and receiving processes. The system does not normally interpret the information given in either the short message or in any accompanying segment, unless the destination process is one of the system processes, such as a device control process. In this case, certain conventions are assumed for the format of messages.

A document to be read by system processes is assumed to consist of a number of records. In the case of a short message sent to a system process, just one record is assumed to exist in the message. Each record consists of a four byte count of the number of data bytes, followed by the data itself. The count is equivalent to a 32 bit integer. This is always in a packed, low order first, form so that the first byte holds the bottom eight bits of the integer, etc. The end of a document is signified by a record with a data count of -1.

15.4 CONNECTION TO DEVICES

Output devices fall into two broad categories. Namely, named devices such as 'LPT' and devices paired with input devices such as a VDU screen. The former can be specified by name (e.g. LPT*) in commands such as DEFINE.OUTPUT (Section 3.3.1) and the latter through the reply mechanism.

Input devices are connected to processes either by means of the connection mechanism of Section 2.1.1 or by means of the following command

1) SET.CONNECTION (ADDR [LOGICAL8], ADDR [LOGICAL8], INTEGER)

This procedure is used to direct input from a device to a process. A process may be the destination of input messages from one or more devices.

P1 is a vector identifying the device to be connected, the source mode and sequence number. The device identification, in the form of a system process number, appears in the first two bytes with the machine number in the second byte. The mode and sequence number are used when sending messages from the device. The suspend process mode does not apply to devices.

P2 is a vector identifying the process to receive input messages from the device. The vector is passed as the destination vector when sending messages,

A process can set the destination of messages from a device under the following conditions:

- a) An unconnected device can be connected
- b) The process currently connected to a device may change the connection
- c) The primary process of a device, as defined in Section 2.1.1, may change the connection.

If P3 is non-zero the primary connection is changed instead of the current one.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 16

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

16 SEMAPHORES

The primary synchronisation mechanism in MUSS is its message system. This allows collaborating processes to be distributed across a network. However, it is possible for processes in the same machine to share segments, providing the memory management hardware allows multiple virtual addresses to be associated with the same physical address. The common segments are one example of this. Also, when the underlying hardware architecture is suitable, two or more processes may open the same file and share one physical copy. Thus some special applications may run as multiprocess systems with the processes communicating through shared store.

The following set of commands is provided to facilitate the synchronisation of these processes.

16.1 COMMANDS

1) CREATE.SEMAPHORE (INTEGER)

This procedure creates a semaphore with the number given by P1. Normally, up to 8 semaphores may be used within each process, and P1 is taken modulo 8.

2) PASS.SEMAPHORE (INTEGER, INTEGER, INTEGER, INTEGER)

This procedure is used to enable a semaphore of the current process to be made available to any process for which the SPN/PID is known. P1 gives the semaphore number in the current process, P2 gives its number in the recipient process and P3 gives the SPN of the recipient process and P4 its PID. When a semaphore has been passed in this way the recipient and creating processes may both use the P and V operations described below.

3) INIT.SEMAPHORE (INTEGER, INTEGER16)

This command may only be issued by the process which created the semaphore P1. It initialises the semaphore value to P2 and sets its wait queue empty.

4) P.SEMAPHORE (INTEGER)

This command performs the Dijkstra “P” operation on the semaphore P1

```
i.e.      value(P1) := value(P1)-1
          WHILE P1 < 0 WAIT
```

Thus the current process may be placed in a waiting state until some other process causes the semaphore value to become ≥ 0 . Any number of processes may be waiting on the same semaphore and they are ordered in arrival order.

5) V.SEMAPHORE (INTEGER)

This command performs the Dijkstra “V” operation on the semaphore P1

```
i.e.      value(P1) := value(P1)+1
          IF P1  $\leq$  0 THEN FREE FIRST WAITING PROCESS.
```

16.2 RELEASING SEMAPHORES

When any process ends which is connected to a semaphore the number of users of the semaphore is decremented by 1. If there are other processes still connected to the semaphore it will remain in operation. When the terminating process is the only process connected to a semaphore it will cease to exist and the data structures associated with it will be released.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 17

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

17 DISC/MAGNETIC TAPE FACILITIES

This Chapter outlines the procedures provided in MUSS for handling exchangeable disc and magnetic tape devices. The facilities described are used by system software such as the floppy disc manager, but are also available to users wishing to access private volumes outside the file system.

The basic system distinguishes between two types of device: fixed and variable block transfer devices respectively. Most exchangeable discs and some magnetic tapes are fixed block devices. These are pre-addressed, and hence a device address can be specified with each transfer request. Most magnetic tapes as well as some exchangeable disc systems, are variable block devices. With these there is no concept of ‘address’ on the device and accessing is usually sequential. For convenience, fixed block devices are referred to as ‘discs’, and variable block devices as ‘tapes’.

Individual tapes and disc cartridges (volumes) are identified to the system by a name and a label. The name is used to identify the volume to the operators – thus it will usually be a serial number enabling the operator to find the required volume upon request. The label is under the control of the user who ‘owns’ a particular volume, and serves two purposes. Firstly, it provides a check that the correct volume, and not another which has somehow acquired the same serial number, is loaded. Secondly, it acts as a ‘password’, protecting the volume against unauthorised access by other users. Labels are written by one of the procedures described below. A facility exists to force the system to accept an incorrectly labelled volume. This enables non-standard volumes and volumes which have not yet been labelled to be accessed. This facility is described with the operator commands in Chapter 8.

In the basic system, deadlock situations are resolved by the operator by dismounting selected volumes. Thus, all processes using discs or magnetic tape should be prepared to receive the response ‘device not available’ to any requests, and to recover as appropriate.

17.1 COMMAND PROCEDURES

The commands associated with disc and tape management fall into two categories, namely the acquisition and control of a disc/tape unit and the

commands for actually driving the unit, performing transfers, etc. The high level control functions are common to both tapes and discs, and a description of these commands immediately follows. The physical characteristics of the two types of media require a different set of functions for performing transfers, and so these commands are described separately for the fixed and variable block devices.

17.1.1 Organisational Functions

1) MOUNT (LOGICAL64, LOGICAL64, ADDR [LOGICAL8])

This procedure requests a particular tape or disc to be mounted on an available unit, and allocated to the calling process. The unit number is returned in PW1. The first parameter is used to signify whether the medium is a tape or disc, and so P1 should take the value "TAPE" or "DISC". P2 is the name of the tape/disc to be loaded. If it is not already available on a drive, then a message is output to the operator to mount it. This message will be repeated periodically until either the tape/disc becomes available or until the operator signals CANT.DO to the process. On some smaller systems, the user is expected to mount his own tape or disc and will not be prompted by MOUNT. Instead, if the tape or disc is not currently mounted, a status will be returned indicating that the medium is not available.

The expected value of the tape label appears in P3. If a tape of the required name is available but P3 does not match the label on the tape, an error is signalled to the process.

2) RELEASE (INTEGER)

This procedure is used to release a disc or tape unit previously allocated by the MOUNT command. The device is disengaged and the operator informed so that the volume may be removed from the drive. P1 specifies the unit number to be relinquished. If this is negative, all disc and tape drives assigned to the current process will be released.

3) WRITE.LABEL (INTEGER, LOGICAL64, ADDR [LOGICAL8])

This procedure changes the name of a volume which has already been mounted. P1 specifies the unit number, as returned from the mount command. P2 and P3 give the new name and label to be written to the volume.

17.1.2 Fixed Block Transfers

1) ED.READ (INTEGER, ADDR, LOGICAL32, LOGICAL32)

This procedure reads data from a disc or pre-addressed fixed block size magnetic tape to a buffer in the current process' virtual store. P1 states the unit number of the required volume. P2 gives the virtual address of the start of the buffer. P3 gives the address of the data on the disc and P4 the amount of data to be transferred. Both the virtual address and size parameters are in byte units. There may be hardware dependent restrictions on the values of P2, P3 and P4.

2) ED.WRITE (INTEGER, ADDR, LOGICAL32, LOGICAL32)

This procedure writes data to a disc or pre-addressed magnetic tape from a buffer in the current process' virtual store. The interpretation of the parameters is exactly the same as for ED.READ.

17.1.3 Variable Block Transfers

1) MT.SKIP (INTEGER, INTEGER, INTEGER)

This procedure skips over a number of blocks of data on the specified device. P1 is the unit number and P2 the direction to be travelled (0 = forwards, 1 = backwards). P3 gives the number of data blocks to be skipped. At least 1 block will always be skipped, even if P3 is zero. PW0 might be set to a fault condition if the device tries to skip over a tape mark or beyond the end of the tape.

2) MT.READ (INTEGER, ADDR, INTEGER, INTEGER)

This procedure reads the next block of data from the specified unit (P1). P2 gives the virtual address in bytes to which the data is to be read and P3 states the maximum size of the transfer. P4 gives the direction in which the medium is to be moved (0 forwards, 1 = backwards). As with MT.SKIP, a fault will be signalled if a tape mark or end of tape is encountered.

The actual size of the transfer (in bytes) is returned in PW1

3) MT.WRITE (INTEGER, ADDR, INTEGER)

This procedure writes a block of data to the specified unit (P1). P2 gives the virtual address in bytes of the start of the data and P3 the number of bytes to be transferred.

4) MT.SKIP.TM (INTEGER, INTEGER, INTEGER)

This procedure skips across data blocks on a medium until a tape mark is encountered. P1 specifies the unit number of the tape and P2 the direction of travel (0 = forwards, 1 = backwards). P3 specifies the number of tape marks to be skipped.

5) MT.WRITE.TM (INTEGER)

This procedure writes a tape mark on the specified device (P1).

6) MT.REWIND (INTEGER)

This procedure rewinds the specified tape so that the transfer heads are positioned at the start of the medium.

17.2 UTILITIES

A number of facilities are provided to allow users to read and write tapes according to certain standard formats. The formats determine

- i) whether labels are present on the tape
- ii) whether MUSS housekeeping information held within input/output documents (as described in Chapter [13](#)) is also stored with the file.

Two conventions for labelling tapes are currently used. The first is according to ANSI standard X3.27, which defines the sequence for volume labels, file labels and tape marks. The second is where labels are omitted, and the only textual information on the tape is that of the actual files. A single tape mark separating each file is the only control information used in this case. For both formats, a double tape mark signifies end of tape.

The utilities for reading/creating tapes have a parameter to specify the format required. This parameter may take the following values:

- 0 Files include MUSS housekeeping information, ANSI standard labels are written to the tape. This is the normal form for transmission of files between MUSS systems.
- 1 Housekeeping information is removed from the files ANSI standard labels are included.
- 2 No labels are written to the tape. Files include MUSS housekeeping information.
- 3 No labels or housekeeping information are written to the tape. This format is often used for transmission of textual files between incompatible systems.

1) WRITE.FILE (INTEGER, ADDR [LOGICAL8], INTEGER,
INTEGER, ADDR [LOGICAL8], ADDR [LOGICAL8],
ADDR [LOGICAL8], INTEGER)

This procedure writes a file P2 to the tape on unit P1. P3 specifies the format to be used for the file. The size of the data blocks on the tape is given by P4. If this is negative or zero, a default size of 1 Kbyte will be used.

Parameters P5 to P8 are only applicable to tapes written in ANSI format, and specify information to be inserted into the header label which precedes the file on the tape. P5 is the sequence number of the file on the tape; P6 is the file generation number; P7 is an expiry date for the file and P8 is an accessibility key. The expiry date is specified as five digits, the first two giving the year and the next three the day within the year. Suitable defaults are supplied for P5 to P8 if specific values are not given.

2) WRITE.TAPE (LOGICAL64, ADDR [LOGICAL8], INTEGER,
INTEGER, INTEGER)

This procedure writes a set of files to a tape. The parameters P1 and P2 specify a tape name and tape label respectively. P3 states the format to be used for the individual files on the tape. If ANSI standard format is chosen, a suitable volume label will be written at the start of the tape before the individual files. P4 specifies a block size to be used for the files, with a default as for WRITE.FILE. P5 indicates whether a number of files already

exist on the tape. Thus, if P5 is non zero, the procedure skips to the current end of the tape before appending the new files.

The procedure prompts for the names of the files to be written. If a ? appears in a filename, the procedure selects a set of files from within the current directory whose names commence with the string immediately preceding the ?. If a ? on its own is typed, the entire directory will be copied to tape. A file name of @ indicates that the tape is complete, and appropriate tape marks are written to the tape.

3) READ.FILE (INTEGER, ADDR [LOGICAL8], INTEGER,
ADDR [LOGICAL8])

This procedure reads a file from the tape mounted on unit P1. P2 gives the filename which will be used for securing the file. P3 specifies the format of the tape. If ANSI format is used and a null filename is specified, a filename will be extracted from the file header label on the tape. P4 is only applicable to ANSI tapes, and should be a vector into which the 80 character file header label is returned.

4) READ.TAPE (LOGICAL64, ADDR [LOGICAL8], INTEGER)

This procedure reads a set of files from a tape. The tape name and label are specified in P1 and P2 respectively. P3 states the format of the tape. If the tape is in ANSI format, the names of the individual files are extracted from the file header labels preceding each file on the tape. For simple textual tapes, the procedure prompts for the individual file names before reading each file. The procedure returns when a double tape mark is encountered.

5) FIND.FILE (INTEGER, ADDR [LOGICAL8])

This procedure searches an ANSI format tape for a file with name P2. P1 gives the unit number of the tape drive.

At the end of the command, the tape is in a position where a subsequent call of READ.FILE will copy the file from the tape. As the search is in a forward direction along the tape starting at the current position, a file will not be found if it has already been passed over. If the user does not know the relative position of the required file on the tape, an MT.REWIND command

should be issued initially to ensure that the complete tape is searched.

6) COPY.TAPE (LOGICAL64, ADDR [LOGICAL8], LOGICAL64,
ADDR [LOGICAL8], INTEGER)

This procedure may be called to create a copy of a tape. P1 and P2 give the name and label of the tape to be copied, and P3 and P4 give the name and label of the tape to be overwritten. The parameter P5 gives the action to be taken on errors. If non-zero, the copying process is halted on detection of an error. When zero, errors are monitored, but an attempt is made to continue the copying process.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 18

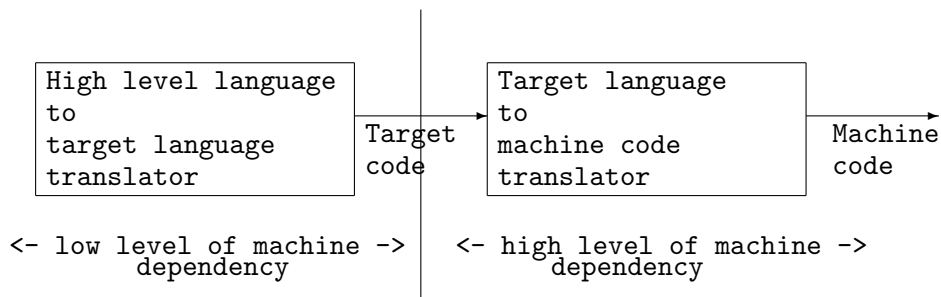
UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

18 COMPILER TARGET LANGUAGE (MUTL)

18.1 INTRODUCTION

The tasks performed by a compiler in translating a high level language program to some object form may be divided into two parts. Some tasks, such as lexical and syntax processing, are largely independent of the machine on which the compiler is running; while other tasks, such as code generation, are very much more dependent on the type of machine that the program is being compiled for. The concept of a compiler target language is to present an abstract machine model to which the compiler targets its code, thereby increasing the proportion of the compiler which is almost machine independent. Thus a compiler may be considered in the two parts shown below



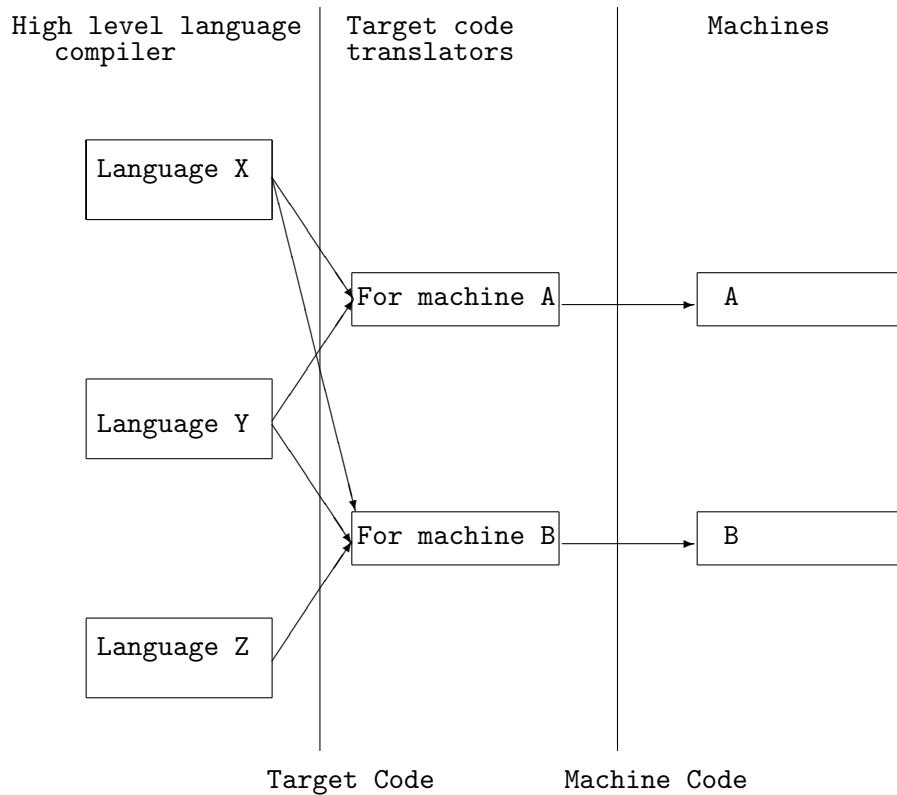
The compilers which run under the MUSS system are organised in this way and the target language is the Manchester University Target Language (MUTL).

The justification for this separation of a compiler is mainly threefold.

1. The abstract machine model can be the target machine for several high level language compilers, thus all code generation tasks of these compilers is centralised, and they are thereby simplified.
2. A single compiler can produce code for several different machine types, by having several abstract machine code translators available to the compiler.

3. On transferring the compilers to another type of computer, only the abstract machine code translator needs be rewritten for the new environment.

The diagram below illustrates points 1 and 2.



Although the main function of a target language is code generation, it may to advantage perform other subsidiary functions. For example, the task of compilation mode control and the provision of high level language run time diagnostics may be functions of the target language.

Experience of the design of several abstract machine models for multi-language multi-machine usage has led to the following conclusions.

- 1) The design of the target model is strongly influenced by the efficiency of the code required. If the best object code efficiency possible for a multiple register machine is to be achieved the compiler must be aware of the number and type of registers available on a particular machine

in order to make optimum use of registers. For single accumulator and stack based machines, abstract models based on a single accumulator or stack architecture are equally effective, and such models are capable of reasonably good object code efficiency for multiple register machines.

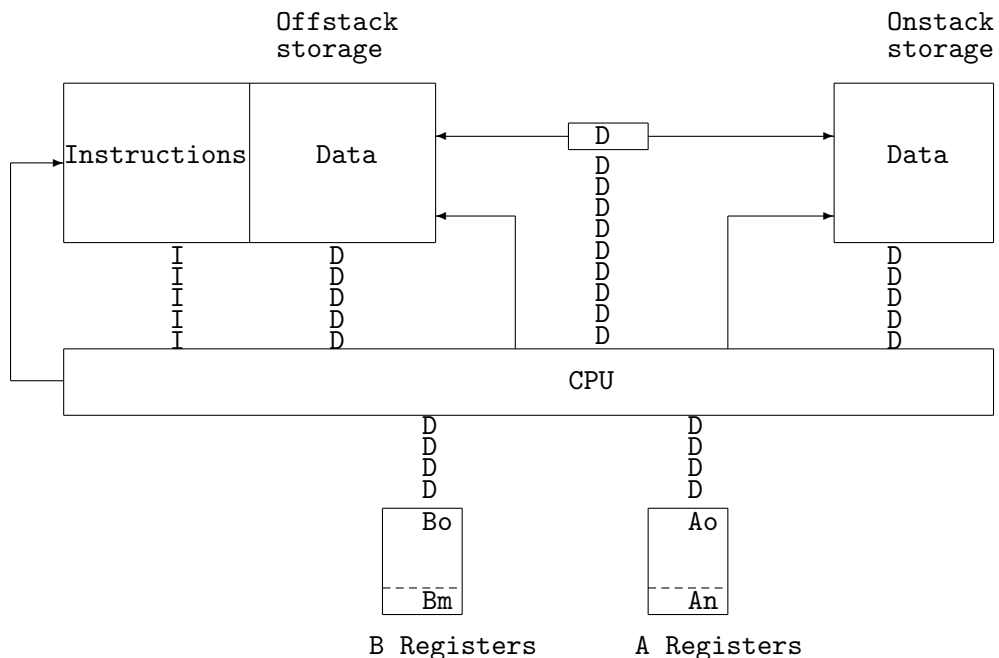
- 2) Optimisations such as removing multiple evaluations of sub-expressions and moving invariant expressions outside loops should be in the machine independent part of the compiler and not in the target language translator. As a consequence of this the unit of code generation of the target language can be in terms of single instructions in the abstract model.
- 3) Compiler target languages which are easily portable normally achieve this by restricting their data types and data structures, consequently these languages often have considerable inefficiency in data storage mapping and inefficient data access code. In order to obtain efficient data access code on different machines, and allow heterogeneous data structures, the target language must be sympathetic to the type and structure of data.
- 4) The MUTL abstract model caters primarily for Fortran, Pascal, Algol and Cobol as well as the system implementation language MUSL. An actual implementation of the model may omit some of the capabilities of the model when they are not required for the set of high level language compilers required at a particular installation. For instance if only Fortran and Algol are required, the target language need not have decimal arithmetic. As further compilers based on MUTL are developed, then where necessary the MUTL model definition will be revised to incorporate new requirements.
- 5) The communication between the compiler and the target language should be essentially one way. Restricting the target language in this way means that the high level language to target code translation and target code to machine code translation may, where necessary, be completely separated. There are several advantages of this approach:
 - a) an encoded form of the target language (MUTL) can be utilised in self bootstrapping the high level language compilers,
 - b) in a computer network, compilation for several different machine

types may be done on a central machine, and if the output of the compiler is in the encoded target language code, it can be assembled into machine code on the appropriate execute job machine in the network, and

- c) for small machines, which have a small address space it provides a natural division of the compiler into two passes.

In the abstract model data is allocated in terms of scalar or vector quantities, and from these basic data components more complex data structures are built. The model provides addressing functions to access such data structures, but the target language provides the declarative functions for the allocation of data in the abstract model. As the target language deals with data at this high level, data accessing on the actual machine can be handled efficiently.

The diagram below illustrates a simplified view of the MUTL model. It basically consists of registers, stores, and a CPU.



18.1.1 Registers

As mentioned briefly in the introduction, optimum register usage on a multiple register machine is best achieved if the compiler can adapt to the number of different registers available. Thus the MUTL model has multiple B registers and multiple A registers, but only a single D register. At present all issued versions of MUTL implementations only cater for one B register and one A register.

The A registers operate in one of several modes:

- integer
- unsigned integer
- floating point
- fixed point decimal
- pointer
- generic

The B registers are primarily for subscription of data structures. They operate in integer and unsigned integer modes.

The D is a data selection register that references into a data structure in order to access a component from it.

18.1.2 Arithmetic precision

To allow MUTL to be implemented on micro, mini and mainframe computers the precision of an A register is selected from the following:

- 1, 8 and 16 bits (with integer mode only)
- 32, 64 and 128 bits (all modes of A except decimal)
- any byte sized multiple up to 80 bits (decimal).

An implementation of MUTL need only implement sufficient arithmetic capability for the high level languages concerned. The permitted precisions for B registers are 8, 16 and 32 bits.

18.1.3 Instruction and Data Stores

The model incorporates off-stack storage and on-stack storage. Instructions are fetched in sequence from the off-stack store and executed in the CPU,

until a control transfer order is executed at which point instruction fetches continue from some other 'labelled' place in the store.

Data is contained in both stores. The off-stack store generally contains data that does not belong to any particular procedure e.g. Fortran Common, and permanent data of a procedure e.g. OWN in Algol, SAVE in Fortran. The on-stack store holds the data of all the currently active procedures. Within the stack frame of a procedure, its data is split into three groups:

- a) parameters of the procedure
- b) variables (scalars, vectors and aggregates)
 local to the procedure
- c) partial results stored in a LIFO basis.

18.1.4 Operand types and lengths

Data in store may be considered as a sequence of simple typed operands, where each operand is either arithmetic or a pointer.

An arithmetic operand is either:

- a) 1 bit long
- b) or between 1 and 16 bytes long
- c) a string of up to 32 bits.

At operand declaration time type is specified. Arithmetic operands may be of real, aligned or unaligned integer and decimal type. A 1 bit operand is always of unsigned integer type, real operands are restricted to precisions of 2, 4, 6 and 16 bytes, and decimal operands are limited to 10 bytes. In addition bit string operands are restricted to signed and unsigned integer types.

Pointer operands refer to: data items, labels and procedures. However, the information content required for such entities differs between high level languages, consequently six types of pointers are distinguished

- a) pointer to a data item
- b) bounded pointer to a data item
- c) pointer to a label
- d) pointer plus environment information of a label
- e) pointer to a procedure
- f) pointer plus environment information of a procedure

18.1.5 The CPU

The instruction set is mainly one address and most instructions consist of a function part and an operand part. It is fully described in Section 18.9. Implementers of MUTL code generators should be aware that whilst many MUTL instructions translate into one machine instruction, and some may translate into several machine instructions, others such as those concerned with operand selection and arithmetic mode selection are not expected to generate any code.

18.2 OVERVIEW OF MUTL CONCEPTS

In addition to providing the imperative functions of code generation for compilers, MUTL provides declarative functions for data items, procedures, labels, etc. Before a specification of the procedural interface is given, it is necessary to introduce briefly; compilation units, storage control, the MUTL concept of types, names, machine code efficiency, diagnostic information and MUTL code output.

18.2.1 Compilation Units (Modules)

The basic unit of compilation is a module which normally contains one or more procedures, and a program or library may involve several such modules. A module contains the specifications for the entities which are considered local to the module. Some of these may be indicated as ‘exports’ in which case they are available for import into other modules. In order that modules may be compiled independently they must contain specifications of their imports, which at compile time are taken on trust. However, these import specifications are passed to MUTL, so that at load time it can check their consistency with the corresponding export specifications. The entities referenced in import or export specifications are called interface entities. Only areas of store, data items, data types, literals, procedures and subroutines, and labels may be interface entities. In fact areas of store are slightly different from other interface entities in that they do not belong to any particular module, but are shared between all modules using them.

It is the symbolic name of the interface entity, specified in characters, that enables inter-module references to be resolved. Therefore, each interface entity of one kind must be uniquely identifiable by its symbolic name.

18.2.2 Storage control

As discussed earlier storage is split into off-stack and on-stack storage. On activation of a dynamic procedure, a new stack frame is created to contain its parameters and local variables. Further space on the stack is obtained as required at run time by the TL.MAKE function. The off-stack store is considered by MUTL as a number of areas into which code is planted and data is statically allocated. There are 255 such areas numbered from 1 upwards, from which current code and data areas are selected as appropriate. The amount of store allocated in an area while selected as a current area is known as a partition. The selection of an area as current defines the start of a partition, and the selection of another area terminates the partition. An area thus consists of one or more partitions.

The procedure TL.S.DECL is used to statically allocate off-stack and on-stack data. A portion of an area (off-stack or on-stack) may be statically allocated as a SPACE so that at run time variables may be dynamically allocated within it, this is achieved by the TL.MAKE function.

The organisation of the stack is the MUTL translators responsibility. However, non-local accesses will generally involve static links, or a display approach, or even a mixture of both methods.

18.2.3 Data types

In MUTL there are a set of predefined data types known as the basic types. Additional types known as aggregate types may be specified. These consist of a concatenation of several basic types or previously defined aggregate types. A variable of aggregate type may, when required, be considered as a single entity. In general aggregate types may be considered as a hierarchic structure of types. For variables of aggregate types, individual fields are identified by their relative position, counting from zero (at each level in the hierarchy). Within an aggregate type, fields may have several alternative type definitions, thus allowing variables to contain variant fields.

Basic types are classified into arithmetic and pointer types. An arithmetic basic type specifies size and mode information (e.g. 16 bit integer) for the data item. Pointer types specify whether the type of the data item referenced is a data, label or procedure item. There is also a typeless data pointer, but whenever a pointer of this kind is loaded into the D register, type information must first be given by calling TL.D.TYPE.

18.2.4 Allocation of names

MUTL names are used to identify and reference many different kinds of entities, for example data variables, literals, procedures, type specifications, etc.. There are two sorts of MUTL entities local and global. Local entities declared within a procedure or block lose their identity at the end of the procedure or block in which they are declared. Global entities lose their identity at the end of the module in which they are declared. In order to maintain the one way communication of MUTL, for reasons specified earlier, MUTL and the compilers use the following simple algorithm for allocating names within a module. Names exist as numbers in the range 2 to 1983. Names for local entities are allocated consecutively from 2 upwards as local entities are declared. At the end of procedures and blocks all names allocated for local entities within the procedure or block become undefined, and they are re-assigned consecutively as new local entities declarations occur. Names for global entities are allocated consecutively from 1983 downwards.

Subject to the non-local accessing restrictions of variables and labels of procedures (discussed later in Section 18.7) all names currently defined can be referenced regardless of the level of procedure nesting at which they are defined.

18.2.5 Code efficiency and optimisation

The optimisation task is shared between compilers and the MUTL translator. It is the compilers job to perform global optimisation techniques, and to produce optimum MUTL code sequences. The MUTL translator then applies peephole type optimisation when generating code from those sequences.

As mentioned earlier, the strategy in utilising multiple registers is to have optimising compilers tune their output code to the number and type of registers available. Thus the procedure TL.ENQ.REG (18.12) supplies this kind of information about the compiling environment. Normally a MUTL translator reserves some actual machine registers for such tasks as stack management, operand conversion and data accessing, the remaining machine registers being nominated as MUTL registers. Furthermore to assist efficient code generation the use of MUTL registers is deterministic, this is achieved with assistance from the compiler.

On some actual machines the intelligent use of base registers reduces significantly the size of operand access code (for example, on the VAX

computer when a base register is loaded to a static area then access to an operand relative to the base register costs 2 bytes, while an absolute operand access costs 5 bytes). However, when the number of base registers is limited then global analysis is necessary in determining an optimal strategy for base registers. Thus there is a dichotomy in that it is the compiler's role to perform global analysis but as the operands in the MUTL interface are at a high level, code generation details of operand accesses are a MUTL function. The solution to this is to have a set of MUTL base registers, and as with A and B registers the number of base registers is dependent on the actual machine and is obtained by calling TL.ENQ.REC. A MUTL base register is loaded either with the address of a MUTL segment or the address of a frame on the stack, this is done under the guidance of the compiler.

To use base registers effectively for accessing non-local operands on the stack the compiler needs to cater for static-link and display based stack organisations. A compiler should not load a base register for accesses within the currently active frame or any frames covered by the display; the procedure TL.ENQ informs the compilers of how many procedural textual levels are covered by the display. In accessing an operand covered by a base register MUTL may then plant an access relative to a base register.

18.2.6 Program error detection and diagnostics

The checking for possible program errors is normally done by a mixture of hardware and software. MUTL normally provides some of the software checking, and control of this checking by compilers may be required. The MUTL procedure TL.CHECK provides this control.

The run time diagnostic procedures in MUTL provide primitives to support machine independent diagnostic software for such tasks as producing compile and data maps, debugging (batch and interactive), and profiling. Therefore source language reference information such as data item symbolic names, procedure symbolic names and source line numbers are passed into MUTL during compilation, this and the relevant declarative information is retained at the end of compilation if run time diagnostic support is required.

18.2.7 MUTL Code Output

The forms of code output produced by MUTL depend largely on the requirements of a particular installation.

A compilation produces either all or part of a program or a library. When compiling a library a directory is automatically generated for the libraries interface. Apart from the directory structure, programs and libraries are similar. The code at the outer textual level of a compiled library (i.e. not embedded in a procedure) is executed, as installation code for the library, whenever it is loaded. The definition of library and program files is achieved by MUTL calling procedures in the Library Organisation section of the MUSS Library

MUTL views a program or library as a collection of MUTL segments, these segments are numbered 0, 1, 2 etc. The model of a program is of MUTL segment 0 containing code while other segments normally contain data, but may contain code. Entry to a program is always via MUTL segment 0, therefore any entry and exit sequence code concerned with running a program is added to segment 0 by MUTL. The model of a library is of MUTL segment zero containing a library entry table, followed by code for the library procedures. There is a directory structure created for the library, but its placement and organisation is dependent on the library organisation for a particular machine. Entry to the library initialisation code is always via MUTL segment 0, therefore, any entry and exit sequence code necessary is added to segment 0 by MUTL.

Many situations arise where it is necessary for the user to exercise control over the mapping of the module areas into actual storage. Examples of this need are in overlaying for machines with limited address space, and the creation of operating systems where strict control of the placement of areas is required. In some high level languages such control is a language feature, while in others, where it is necessary, control is provided either by compiler directives or by job commands. The procedures in [18.4](#) enable the compiler to control the placement of compilation output

18.3 TYPE DEFINITIONS

The procedures in this section enable aggregate types to be defined and a MUTL name allocated. In other contexts where the type of an entity has to be given, a MUTL TYPE SPECIFIER is used. This is an encoded 14 bit integer whose least significant two bits indicate whether the specifier relates to

- 0 an entity of the given type
- 1 a pointer to an entity of the given type
- 3 a bounded pointer to a vector of entities of given type.

The other part of TYPE SPECIFIER either defines a basic type or the MUTL name of a user defined type. Basic types have values which are either less than 64 or greater than 2047 (shifted left two places to accommodate the two bits defined above). Thus in order that they may be distinguished, user types are actually represented by their MUTL name + 64 (again shifted left two places). Thus a TYPE SPECIFIER for basic types whose size is specified as one or more bytes is encoded as:

Bits 2–5 specify the size of the type as the number of bytes less one (e.g. 1 means a 2 byte type)

Bits 6–7 specify the type mode.

- 0 – real
- 1 – signed integer
- 2 – unsigned integer
- 3 – signed fixed point decimal with a representation of a decimal digit per nibble with one nibble for the sign. Thus a 7 byte decimal represents a signed 13 decimal digit number.

Bits 8-13 0

As real types are restricted to 2, 4, 8 and 16 bytes, the above leaves 48 spare encodings of bits 2 to 7. Some of these are utilised as follows:

%20 – denotes a 1 bit scalar

%01 – Typeless unbounded data pointer

%03 – Typeless bounded data pointer.

%24 – Pointer to a procedure.

%28 – Pointer to a procedure and its environment.

%2C – Pointer to a label.

%30 – Pointer to a label and its environment.

A TYPE.SPECIFIER for basic types whose size is specified as a bit string

is distinguished by bit 13 being a 1 the rest of the encoding being encoded as

Bits 2–5 0

Bits 6–7 specify the type mode

- 1 – signed integer
- 2 – unsigned integer

Bits 8–12 specify the size of the type as the number of bits less one (e.g. 2 means a 3 bit type).

Bit 13 1

1) TL.TYPE (ADDR [LOGICAL8], INTEGER)

This starts the specification of an aggregate type and allocates a MUTL name for the type, unless one had previously been allocated for it. Type specifications may not be nested, but an aggregate type may include previously defined types and pointers to types which are not yet fully specified. The constituent fields of the aggregate are added by calling TL.TYPE.COMP repeatedly. During the specification of a type only calls to the MUTL procedures TL.TYPE.COMP, TL.END.TYPE and TL procedures to define the current literal are permitted.

The second parameter specifies whether the type is internal to the module, an export from the module, or an assumed type from another module. A type may be imported without its type detail being re-specified in this module in which case there are no further calls to TL.TYPE.COMP or TL.END.TYPE: this situation arises when the language compiler need not be aware of the type details in compiling a module.

An *incremental* type is an aggregate type where the type specification is given incrementally over several modules. For an incremental type a skeleton specification is first given for the type so that MUTL knows its overall size for code generation purposes. The actual specification is built up incrementally by modules adding fields to the type. If a module adds fields to an incremental type it cannot access directly any of the other fields added by other modules.

Note that a type specification containing no fields is permitted.

The position of a component, counting from zero, within the type definition is used in accessing it.

Parameters:—

- | | |
|-----------|--|
| P1 | Bounded pointer to a byte vector containing the symbolic name in characters of the type. When the type is an interface entity the symbolic name identifies the type, it is also retained for run time diagnostic purposes. |
| P2 | Bit 14 = 1 type exported from module. Bit 15 = 1 type of an assumed entity to be imported. |
| Bits 0-13 | <p>%0000 means a MUTL name is to be allocated and its type specification follows immediately.</p> <p>%0001 is as %0000 except that the type has no alternatives and contains only one field. In referencing this type selection of the contained field is automatic, i.e. no SEL FLD is necessary, such a type is known as an automatic type. This feature enables compilers to create extra types to assist in type checking and for defining enumeration types.</p> <p>%1000 means a MUTL name is to be allocated and its type specification is not specified in this module; MUTL obtains the specification from the export of the type from another module.</p> <p>%2000 means a MUTL name is to be allocated but its type specification is given later in the module.</p> <p>%3000 indicates fields are to be added to an incremental type. A MUTL name is allocated and further fields are added to the type's specification by calling TL.TYPE.COMP to add each new field, and then finally calling TL.END.TYPE. Within the module the newly added fields are referenced by ascending field numbers 0, 1, 2, etc. regardless of how many fields there are already in the type.</p> <p>%0002 to %07FF means the type specification now follows for the MUTL name specified in bits 0-10, which had previously been allocated with %2000 specified for this field</p> <p>%3002 to %37FF is as %0002 to %07FF except that fields are to be added to an incremental type as in %3000 above.</p> |

Example:

To create a type T consisting of a vector of two integers (4 bytes each) TVI, and a vector of three reals (8 bytes each) TVR.

```
TL.TYPE(["T"],0)
TL.TYPE.COMP(%4C,2,["TVI"])
TL.TYPE.COMP(%1C,3,["TVR"])
TL.END.TYPE(0)
```

A type S consisting of an integer (4 bytes) SI, and a ten element vector SVT where each element is of type T can then be created by:

```
TL.TYPE(["S"],0)
TL.TYPE.COMP(%4C,0,["SI"])
TL.TYPE.COMP(MUTL NAME of T*4+256,10,["SVT"])
TL.END.TYPE(0)
```

The ten element vector is referenced as the second component of S (i.e. SEL FLD 1).

2) TL.TYPE.COMP (INTEGER, ADDR, ADDR [LOGICAL8])

Adds the next component to the current aggregate type specification. The first two parameters specify the type of the component.

Parameters:–

- | | |
|----|--|
| P1 | Bits 0-13 give a TYPE SPECIFIER as described above at the start of 18.3 .
Bit 14 a value of one means that P3 specifies a list of names for an enumeration type. |
| P2 | 0 :scalar
> 0:vector, parameter specifies number of elements.
-2048:vector, current literal specifies number of elements.
Other negative values: vector, the parameter is the negated MUTL name of a literal whose value gives the vectors dimension. |

P3 Bounded pointer to a byte vector containing the symbolic name in characters of the component, or a list of symbolic names for the names of an enumeration type (each name in the list is terminated by a zero valued byte). This parameter is for run time diagnostic purposes only.

3) TL.END.TYPE (INTEGER)

Terminates the addition of fields to the aggregate type currently being defined. P1 indicates the end of a type specification, the end of an alternative within the type, or that the type specification is the skeleton of an incremental type.

Parameters:—

- P1 0 — end of type specification, or end of fields being added by this module to an incremental type.
- P1 1 — another alternative type specifications follows
- P1 2 — indicates a skeleton specification for an incremental type.

18.4 STORAGE MAPPING

As previously explained in [18.2](#) MUTL plants code and literals within the current code area and allocates statically mapped storage within the current data area. All areas are placed in MUTL segments. The procedure TL.SEG introduces a MUTL segment into the compilation, while the procedure TL.LOAD specifies into which MUTL segment a particular area is placed. Thus a module's areas may be mapped to several MUTL segments and a MUTL segment may contain areas from several modules. The characteristics of a MUTL segment are that it occupies, when in store at run time, a contiguous address space and all parts of it have the same level of protection (e.g. same level of protection and overlay type) and similar items throughout the space are accessed in an identical manner. Note that if there is more than one area mapped to a MUTL segment the partitions of an area may not be allocated continuously within the MUTL segment, and generally the currently selected data and code areas should not be mapped to the same segment as interleaving may occur of code and data. Normally each data declaration can be considered as unrelated from other data declarations, but for off stack data involved in a Fortran type EQUIVALENCE it implies a

specific ordering of the data. Therefore the TL.EQUIV.POS is provided to permit the declaration point of a variable to be specified explicitly within an off-stack data area.

1) TL.SEG (INTEGER, ADDR, LOGICAL32, ADDR, INTEGER)

Introduces a segment into the compilation. These are known as MUTL segments and have numbers in the range 0 to 31. On introducing MUTL segment 0 initialisation code concerned with compiling a program or library is planted automatically by MUTL.

Parameters:—

- | | |
|----|---|
| P1 | Number of MUTL segment. |
| P2 | Size of MUTL segment in bytes. On machines where control transfers between virtual store segments is severely restricted (e.g. procedure calls only) the maximum size of a MUTL segment is normally limited to that of a virtual store segment, whereas for other machines a MUTL segment may span several consecutive virtual store segments. A size of zero means a segment of default size is created. |
| P3 | Runtime address of MUTL segment specified in bytes. A value of -1 means that the segment's address will be allocated automatically. |
| P4 | Compile time address of MUTL segment specified in bytes. A non negative value specifies the compile time address for the MUTL segment. A value of -1 means the compile time address will be automatically allocated; a value of -2 means that the MUTL segment need not be allocated at compile time as no code production or static data initialisation occur for this MUTL segment (in order to retain compatibility with earlier versions of MUSS a value of -3 is permitted and this is equivalent to P4 = -2 and bit 5 of P5 set). |

P5 This parameter specifies the properties of the MUTL segment.
 Bit 1 = 1 means that the MUTL segment at run time requires
 execute access.
 Bit 2 = 1 as Bit 1 but for read access.
 Bit 3 = 1 as Bit 1 but for write access.
 Bit 5 = 1 The MUTL segment will not be loaded automati-
 cally when the library or program is loaded. The compile
 time image if created, of the MUTL segment is not released
 at the end of the compilation.

2) TL.LOAD (INTEGER, INTEGER).

This procedure specifies into which MUTL segment the partitions of an area will be mapped. If the MUTL segment has not been defined a MUTL segment is created with default parameters i.e. TL.SEG (SEG.NUMBER,0,-1,-1,0). The mapping of an area must be given before the area is used.

When modules are bound together then once an area segment mapping is established it applies to all modules following until a new mapping is given.

Parameters:—

P1 Segment number.
P2 Area number.

3) TL.CODE.AREA (INTEGER)

Nominates the current code area and sets the current position to the next available location within it.

Parameters:—

P1 Area number.

4) TL.DATA.AREA (INTEGER)

This procedure nominates the area specified as the current data area and sets the current position within the area. Statically allocated variables are mapped to the current position in the currently selected data area.

Parameters:–

- P1
- > 0 Area number
 - 0 stack frame of the current procedure
 - < 0 stack frame of procedure whose textual level is
[current procedural textual level - P1].

5) TL.COMMON (ADDR [LOGICAL8], INTEGER32, INTEGER)

This procedure associates a symbolic name with a data area, such an area is termed a ‘common area’. To declare variables in a common area, TL.COMMON is first called, then calls on TL.S.DECL declare variables, finally a call on TL.END.COMMON indicates the end of a declaration set within the common area. In a compilation, sets of declarations can be given several times for the same common area. Storage is allocated from the start of the named data area each time.

The first call of TL.COMMON for a particular common area defines its mapping. This is either explicit or performed by MUTL.

The total storage requirement for a common area is normally defined by the first declaration set for that common. All other declaration sets for the common must not exceed this storage limit. To override this restriction the common area may be nominated as a flexible common area.

Parameters:–

- P1
- Bounded pointer to a byte vector containing the symbolic name in characters of the common area. An unnamed common area is allowed; this is specified by a nil pointer.
- P2
- Size of common area for inflexible common areas.

- P3 Bit 15 = 0 Common area is flexible.
 Bit 14 = 1 TL.COMMON specifies mapping information only,
 i.e. there are no immediately following calls to TL.S.DECL
 or TL.END.COMMON.
 Bit 0 = 13 On the first call of TL.COMMON for a particu-
 lar common area a non-zero value gives the number of the
 data area to which the common is mapped; If the value is
 zero, MUTL maps the common area itself. For subsequent
 TL.COMMON calls for a common area this value must be
 zero.

For some MUTL implementations flexible common areas involve significant overheads, in which case flexible usage is normally restricted to the unnamed common area.

6) TL.END.COMMON ()

This procedure ends a declaration set for a common area.

7) TL.EQUIV.POS (ADDR)

This procedure resets the current position of the current area to the position specified. The data area to which it applies must be off-stack.

Parameter:—

- P1 Position from start of area specified in bytes.

18.5 STATIC DATA STORAGE DECLARATION

Variables declared at the outer level of a module are available to all procedures within the module. For variables declared within a procedure, the kind of procedure determines whether its variables may be accessed non-locally

1) TL.SPACE (ADDR)

Space is statically allocated in the current data area, which could be either on-stack or off-stack, and an integer variable is declared to keep track of the run time usage of the allocated space. A MUTL name is allocated for the SPACE and this name can be used in imperative functions (e.g. TL.PL) to control usage of the space. Structures are created dynamically in the

space by the TL.MAKE procedure.

Parameters:—

- | | |
|----|---|
| P1 | When positive it specifies the size in bytes of the SPACE.
When negative it is the negated MUTL name of a literal
whose value gives the size of the SPACE in bytes. |
|----|---|

2) TL.S.DECL (ADDR [LOGICAL8], INTEGER, ADDR)

This procedure declares a variable which may be a scalar or a vector of basic or aggregate type. A MUTL name is allocated for the variable. For variables of a known dimension then, unless the declaration is for an assumed interface entity of another module, storage is allocated for it within the currently selected data area. For variables of an unknown dimension then it is the set of values that statically initialise the variable that determines its size.

Parameters:—

- | | |
|----|--|
| P1 | Bounded pointer to a byte vector containing the variable's symbolic name in characters. It is used to identify the interface entity and for run time diagnostic identification. |
| P2 | <p>Bits 0 – 13 is a TYPE SPECIFIER, see 18.3 for its encoding unless Bits 14, 15 = 3.</p> <p>Bit 14, 15 = 1 Data item to be exported as an interface entity.</p> <p>Bit 14, 15 = 2 Declaration of an assumed interface entity imported another module.</p> <p>Bit 14, 15 = 3 Data item to be exported as an interface but it is already declared as an import in this module.</p> <p>Bits 0 – 11 give its MUTL name.</p> |

- P3 = 0: Scalar
 > 0: Vector, dimension defined explicitly
 – 1: Dimension of initialised vector unknown. Value initialisation determines actual dimension. Storage is allocated at initialisation time.
 -2048: Vector, dimension defined by current literal
 -4096: Initialised scalar with storage allocated at initialisation time.
 Other negative values: vector, parameter is the negated MUTL name of a literal whose value gives the vectors dimension.
- 3) TL.V.DECL (ADDR [LOGICAL8], LOGICAL32, INTEGER, INTEGER, INTEGER, ADDR)

This procedure enables MUSL V-store variables to be defined and MUTL names allocated to them. After declaration they may then be treated by MUSL as ordinary variables in imperative statements. A V-store variable is a special control/status register of the MUSS ideal machine (refer to MUSS Technical Documentation VOL. 1) that can in general be read or written. If the real machine has an actual control/status that is exactly equivalent, the V-store variable may be mapped directly to the actual register. If the correspondence is not exact then the V-store is mapped onto an ordinary memory location. In this case it may be necessary to update the memory location before a read access and to propagate the implied actions after a write access. Therefore, associated with each V-store of the ideal machine there is a memory address, and optionally a procedure (PRE.PROC) to be called before a read access, and optionally a procedure (POST.PROC) to be called after a write access.

The read and write subroutines associated with V-store variables which are vectors generally need to know the index number of the element to be acted upon. Therefore, on entry to such procedures MUTL makes the index number available. Within the subroutine the index number is obtained by specifying an operand of %1002, known as V.SUB, in the operand description of TL.PL.

In the implementation of index number passing, whenever the actual machine architecture permits it, the index should be passed in the register that corresponds to the B register in the MUTL model, and an implementer

of MUTL may assume that any B register use within a PRE.PROC or POST.PROC destroys the passed index value.

Like other data variables V-store variables may be interface entities.

Parameters:—

- | | |
|----|--|
| P1 | Bounded pointer to a byte vector containing the variables symbolic name in characters. It is used solely to identify the interface entity. |
| P2 | Actual address associated with V-store variable. This may either be a previously declared variable or an address. Thus the parameter may either be the MUTL name of a variable or a literal, or a zero which means the current literal gives the address. |
| P3 | MUTL name of the subroutine to be called before a read access. A value of zero means no subroutine call on a read access. |
| P4 | MUTL name of the subroutine to be called after a write access. A value of zero means no subroutine call on a write access. |
| P5 | <p>Bits 0-13 is a TYPE SPECIFIER, see 18.3 for details.</p> <p>Bit 14=1 V-store variable to be exported as an interface entity.</p> <p>Bit 15=1 V-store variable to be imported as an interface entity.</p> |
| P6 | <p>= 0: V-store is a scalar variable.</p> <p>> 0: V-store is a vector variable, the parameter specifies the dimension.</p> <p>= -1: V-store is a vector variable but its dimension is unknown, this is only permitted on imports.</p> <p>= -2048: V-store is a vector variable, its dimension is given by the current literal.</p> <p>Other negative values: V-store is a vector variable, the parameter is the negated MUTL name of a literal whose value is the vector's dimension.</p> |

4) TL.MAKE (INTEGER, INTEGER, ADDR)

This procedure plants code to allocate at run time a variable at the top of the current stack frame, or on the parameter stack, or in a previously declared space. For the latter the integer control variable of the SPACE denoted by P1 is updated to the next available storage location. For variables of static type then the A register is loaded with a pointer to the variable, while for variables of dynamic types the address is loaded into the A register.

Parameters:—

- | | |
|----|--|
| P1 | The MUTL name of a SPACE (i.e. SPACE > 1) or zero means allocate at the top of the stack frame, or 1 means allocate on the parameter stack. |
| P2 | TYPE SPECIFIER of statically typed variable, see 18.3 for its encoding: 0 means dynamically typed variable in which case the B register contains the size of the type and the A register the alignment of the dynamic type in bytes. |
| P3 | A value of zero means make a scalar variable and yield in the A register an unbounded pointer to it. A non zero value means make a vector variable and yield in the A register a bounded pointer to it. A positive P3 specifies the number of elements that the vector is to contain, whereas a value of -1 means that the value of the B register on ‘making’ the vector determines the number of elements in the vector. For dynamic types this parameter is zero. |

5) TL.SELECT.VAR ()

This procedure notionally declares a SELECT variable and allocates a name for it. A SELECT variable is used to save data structure selection information held in the D register for later use. This variable may only be used in D= and D=> instructions. In a selection instruction sequence involving SEL EL, SEL FLD and SEL ALT, a ‘D=> SELECT variable’ saves the current selection information of D. A ‘D= SELECT variable’ reloads into the D register the previously saved selection information.

Before a ‘D= SELECT variable’ instruction is planted in the code a corresponding D=> to the same SELECT variable must have already been

planted. During run time of the code on executing a 'D=SELECT variable' instruction, the SELECT variable must have previously been assigned by the immediately preceding D=> to the same SELECT variable in the code; in other words, information in a SELECT variable has a static scope.

6) TL.SELECT.FIELD (INTEGER, INTEGER, INTEGER)

This procedure allocates a MUTL name for a field within a data structure pointed to by P1.

Parameters:—

- | | |
|----|--|
| P1 | MUTL name of a Select variable which specifies the containing data structure of the required field. |
| P2 | If the containing data structure is of union type, this parameter specifies the alternative, counting from zero, required. |
| P3 | Number, counting from zero, of field required. |

7) TL.SET.TYPE (INTEGER, INTEGER)

This procedure is called to supply information about an entity (variable or parameter) which was not available at entity declaration time. The procedure is used as follows:

- | | |
|----|---|
| 1) | A variable or parameter may be declared to be a pointer to an unknown data type, this procedure is called to set the data type. |
| 2) | To reset the data type of a 64 bit logical variable or parameter. |

Parameters:—

- | | |
|----|-------------------------------------|
| P1 | MUTL name of variable or parameter. |
|----|-------------------------------------|

P2 If bits 0-13 are non-zero then for typeless pointer's TYPE is as follows

Bits 0, 1	1 Scalar instance of a type
	3 Vector instance of a type
Bits 2-13	encoded as in paragraph 18.3.

and for 64 bit logical variables and parameters, this is restricted to basic types of a size not greater than 8 bytes and pointer types. Bits 0-13 contain a TYPE SPECIFIER, see paragraph 18.3 for its encoding.

If bits 0-13 are zero then bits 14-15 respecify the interface attributes of the variable.

Bit 14,15 = 0 Variable not an interface

Bit 14 = 1 Variable is an export

Bit 15 = 1 Variable is an import.

Respecification of interface attributes is restricted to promotion of an import to an export and removing the import attribute.

8) TL.ASS (INTEGER, INTEGER)

This procedure is used to nominate a variable or a user defined type for the current user defined type literal to which value assignment procedures refer. During value assignment there is the concept of a current assignment position. Normally value assignments start at the first component of the item, but any point can be selected as the current assignment position by calling TL.ASS.ADV appropriately.

If the variable was declared in TL.S.DECL as a vector of unknown dimension, then storage is allocated for the vector when initialisation commences, and the size of the vector is then determined by the set of values assigned. For such variables MUTL assumes that complete value assignment occurs at the same time.

Parameters:—

P1 If bit 14 is zero then bit 0-11 is a MUTL name of an off-stack data variable. If bit 14 is one then bits 0-13 is a TYPE SPECIFIER, see 18.3 for its encoding, of the current user defined type literal.

P2 This specifies the area number in which to allocate storage for vectors of unknown size. A value of -1 means allocate in current code area, and -2 means allocate in current data area.

9) TL.ASS.VALUE (INTEGER, INTEGER)

Assigns the values as specified by P1 to the components starting at the current assignment position within the literal or variable being assigned to, and advances the current assignment position accordingly. The second parameter allows this to be repeated.

Normally P1 represents a single value, the exception to this is when the components being assigned to have a size of one byte, in which case P1 can specify the current basic type literal which had previously been defined to represent multiple byte size values (See Section 18.6).

Parameters:—

P1 The type of name permitted is dictated by the type of the current component having its value assigned.
For components of arithmetic basic type, the name is that of a previously defined literal, and $P1 = 0$ or 1 means the current basic type literal. It is the type of the literal that dictates how the component is assigned a value. This means that local literals are right justified within the component, with zero padding and truncated where necessary; similarly for integer literals, but with sign extension instead of zero padding; and for real literals, generally, these are left justified with zero padding and truncated where necessary. For components of label type, the name is that of a label, or the current literal with a null label pointer value. For components of procedure type, the name is that of an actual procedure or the current literal with a null procedure pointer value.
For components of pointer type, the name is that of a statically allocated data item, in which case the assigned value is a pointer to the data item, or the current literal with a null data pointer.

P2 Number of times the values associated with the first parameter are assigned. If this parameter is positive the parameter itself is the repeat count, else a negative parameter indicates that the repeat count is that of a named literal in which case the parameter is the negated MUTL name.

10) TL.ASS.END ().

This procedure is called to indicate the end of assignments.

11) TL.ASS.ADV (INTEGER).

This procedure advances the current assignment position over the specified number of basic type components of the literal or variable being initialised.

Parameter:—

P1 Number of basic type components to be advanced over.

18.6 LITERAL DECLARATIONS

The procedures in this section provide for the declaration of literals and the compile time evaluation of literal expressions of integer type. To allocate a MUTL name for a literal a value must first be assigned to the current literal, then a MUTL name is assigned. This two stage approach is necessary because the current literal is used in other contexts. Literals of basic type and user defined type are defined in this way. There are two current literals, one of basic type and one of user defined type.

1) TL.C.LIT.16 (INTEGER, INTEGER16)

2) TL.C.LIT.32 (INTEGER, INTEGER32)

3) TL.C.LIT.64 (INTEGER, LOGICAL64)

4) TL.C.LIT.128 (INTEGER, REAL128)

These procedures are used to define the value of the current basic type literal. The first parameter specifies the type of the current literal. The size of the basic type value can be smaller than the size of the value parameter,

in which case the value is right justified within the parameter.

Parameter:–

P1 Bits 0-7 specify a Basic arithmetic scalar type for the value. These are encoded as bits 2-7 of basic type specifications. For TL.C.LIT.64 a data pointer literal may also be defined by setting P1 to the appropriate pointer type. In this case the least significant 32-bits specify the byte address of the pointer, and for bounded pointers the most significant 32-bits specify the dimension.

P2 Value of current literal

5) TL.C.LIT.S (INTEGER, ADDR [LOGICAL8])

This procedure provides an alternative way of defining the current basic type literal value. To expedite the initialisation of byte vectors, this procedure also permits the ‘current literal’ to be a sequence of byte sized values.

Parameters:–

P1 See first parameter of above procedures TL.C.LIT.16, etc... In addition if bit 1 is set then the bytes from the value are taken in sequence and arranged so that a stored value would have these bytes in an ascending address sequence.

P2 A bounded pointer to a byte vector. If the literal type does not have a size of one byte the current literal consists of a single VALUE. In which case the size in bytes of the literal type and P1 must be the same. Otherwise the length in bytes of P1 determines the number of byte sized values associated with the current literal.

6) TL.C.NULL (INTEGER)

This procedure assigns a ‘null’ pointer value to the current basic type literal.

Parameters:–

P1 Specification of a pointer type, see [8.3](#) for its encoding.

7) TL.C.TYPE (INTEGER, INTEGER, INTEGER)

This procedure defines the current literal whose type is specified by P1 with information about a data type. Zero in P2 sets the current literal to the size in bytes and a one in P2 sets it to the boundary alignment in bytes of a data type, whose TYPE SPECIFIER is given by P3.

Parameters:–

P1	See 18.3 for its encoding.
P2	Specifies information required.
P3	See 18.3 for its encoding.

8) TL.C.DATA (INTEGER, INTEGER, ADDR)

This procedure defines the current literal whose value type is given by P1 with a value which is the address of a static data structure. If the data structure is a vector then the address of a particular element may be obtained.

Parameters:–

P1	See 18.3 for its encoding.
P2	MUTL name of a static data structure.
P3	> 0: Element number of vector -2048: Element number specified by current literal.

9) TL.LIT (ADDR [LOGICAL8], INTEGER)

This procedure allocates a MUTL name for a literal which has the value and type of the current literal. For literals of basic type then the above procedures 1) to 8) specify the literal value. For literals of user defined type then the TL.ASS procedures 9) to 11) specify the literal value.

For imported literals the value is optional. When the value is present it is checked against the exported literal value.

Parameters:–

- P1 Bounded pointer to a byte vector containing the literal symbolic name in characters. This is also used if the literal is an interface entity.
- P2 Bit 14 = 1 Literal to be exported as an interface entity.
Bit 15 = 1 Literal to be imported as an interface entity.
Bits 0-13 A value of zero means it is a basic type literal and a value of 2 means it is a user defined type literal, in both these cases the literal value is that of the appropriate current literal. For imported literals any other value is interpreted as a TYPE SPECIFIER, see [18.3](#) for its encoding, of the literal and no value is given.

10) TL.CALC (INTEGER, INTEGER)

This procedure provides compile time evaluation of literal expression within MUTL. Expressions are evaluated in 32 bit integer precision in a literal accumulator. Operands in the expression must be of integer or logical type. A stack of literal values is available for holding intermediate values of an expression.

P1 Operation codes are a subset of the integer operation codes allowed in the procedure TL.PL and are as follows:

0	=>
1	= -
2	=
3	- =
4	&
5	V
6	<<
7	>>
8	+
9	-
10	- :
11	*
12	/
13	/ :
14	==
15	COMP
71	STACK

See description of TL.PL procedure ([18.10.1](#)) for meaning of operation code symbols.

The STACK operation code pushes the value of the literal accumulator onto the literal stack.

P2 Operands are a subset of the operand parameter of the TL.PL procedure. Permitted operands are:
 0 Current literal
 > 1 < 2048 MUTL name of literal
 %1003 Pop the literal value from the top of the literal stack.

For the next six operands a pair of literal operands are popped from the literal stack and one of them, depending on T, is loaded into the literal accumulator and the other is discarded. Let TRUE and FALSE represent these two values, TRUE is pushed onto stack before FALSE.

%1008 ACC = TRUE if T implies =, otherwise ACC = FALSE
 %1009 ACC = TRUE if T implies / =, otherwise ACC = FALSE
 %100A ACC = TRUE if T implies >=, otherwise ACC = FALSE
 %100B ACC = TRUE if T implies >, otherwise ACC = FALSE
 %100C ACC = TRUE if T implies <=, otherwise ACC = FALSE
 %100D ACC = TRUE if T implies <, otherwise ACC = FALSE

Operand restrictions:

- 1) The => opcode has the current literal as it's operand.

18.7 PROGRAM STRUCTURE DECLARATIONS

There are basically three types of program structures; namely procedures, subroutines and blocks.

A *procedure* may have parameters and a result. Procedures are further classified with regards to the permitted non-local availability of their variables, and also whether a procedure is static or potentially recursive. The local namespace is created dynamically on entry to a potentially recursive procedure. The non-local accessibility of a procedure's variables is at one of the following levels.

- I) No non-local variable access.
- II) Non-Local variable access permitted.

If the procedure kind is not specified explicitly, then by default its variables may not be referenced non-locally.

A *subroutine* has no parameters and no stack frame of its own, but it may have a result and be called recursively. For subroutines that are only invoked from the code of the immediately enclosing procedure the code of the subroutine may access (as local variables) the variables of the enclosing procedure, for all other subroutines such access is prohibited.

The same set of procedures is used to define procedures and subroutines.

A *block* is a delimited sequence of code. The block has no name and is not referenced from control instructions. Any labels or variables declared within the block are only accessible within the block, but the variables of the enclosing procedure, if there is one, are still accessible as locals from within the block.

1) TL.PROC.SPEC (ADDR [LOGICAL8], INTEGER)

A specification must be given before a procedure (or subroutine) is either defined or referenced. A procedure (or subroutine) specification consists of one call to TL.PROC.SPEC, followed immediately by the appropriate number of calls to TL.PROC.PARAM to specify its parameters, and terminates with a call to TL.PROC.RESULT which indicates the end of the parameters and also specifies the result type. Of course for a subroutine specification there are no calls to TL.PROC.PARAM.

A procedure (or subroutine) definition starts with a call to TL.PROC. For procedures its kind is specified, unless the default kind of procedure is required, by calling TL.PROC.KIND before the first imperative instruction of the procedure is planted. Finally, TL.PROC.END is called at the end of the procedure (or subroutine).

The symbolic names of parameters, which are used for diagnostic purposes only, are supplied to MUTL by calling TL.PARAM.NAME during the procedure definition, i.e. in between calls of TL.PROC and TL.END.PROC.

This procedure normally allocates a name for the declared procedure (or subroutine).

Parameters:—

- P1 Bounded pointer to a byte vector containing the characters of the procedure's (or subroutine's) symbolic name. It is used to identify the procedure (or subroutine) if it is an interface entity, for runtime diagnostic identification, and for constructing a directory of a library if this procedure is part of its interface. To permit abbreviated names to be used to identify library procedures, the name supplied to MUTL contains both lower and upper case characters. The abbreviated name consists of these upper case characters. The only use of this abbreviated name is in constructing the directory of the library. and for interface and diagnostic purposes the name is handled as if it were all upper case characters.
- P2 Bit 0 A value of zero means the procedure is dynamic i.e. is potentially recursive, and a one means it is static.
Bit 1 A value of zero means this is a procedure specification, and a one means it is a subroutine specification.
Bit 2 A value of one means this is a procedure specification of a built-in procedure. These procedures are predefined to MUTL and may vary between installations. They are provided to improve performance. MUTL normally generates an inline code sequence for built-in procedure calls.
Bit 3 A value of one means this is a library procedure specification.
Bits 6-10 for non-interface procedures this specifies the textual level number for the procedure. The outermost procedure textual level in a module is numbered one. For each nested level of procedure the textual level is increased by one. A value of zero means the procedure textual level is set from the currently active textual level.

Bits 11, 12 and 13 specify whether a MUTL name is allocated, and what specification details are supplied.

0 – A MUTL name is allocated and a specification is given.

1 – A MUTL name already exists for the procedure, but it does not have a specification, in this case bits 0-10 give the MUTL name, and calls to TL.PROC.PARAM and TL.PROC.RESULT will follow to give its specification.

2 – A MUTL name is required but the procedure's symbolic name, and its parameter and result specification is to be supplied later or the name is a multiple entry point to a procedure.

3 – A MUTL name, given by bits 0-10, already exists for the procedure. The symbolic name is given but the rest of the specification of the procedure is supplied later.

Bit 11 A value of one means this is a library procedure specification, note this is an alternative to setting bit 3.

Bit 14 A value of one means the procedure (or subroutine) is an interface entity that is to be exported from the current module.

Bit 15 A value of one means this is a specification of an assumed interface procedure (or subroutine) of another module.

To reference a procedure from a library then its specification must be given. Bits 3 and 15 of P2 are set to one to indicate this.

When compiling a library bits 3 (or 11) and 14 are normally set to one to indicate that this procedure is in the library's interface.

2) TL.PROC.PARAM (INTEGER, ADDR)

This procedure defines the type of the next parameter in the current procedure declaration. A MUTL name is not allocated for the parameter by this procedure. Names for a procedures parameters are allocated by TL.PROC when it is called to commence the definition of the procedure.

Parameters:–

P1 TYPE SPECIFIER of parameter, see paragraph [18.3](#) for its encoding.

P2 = 0: scalar parameter
 > 0: parameter is a vector containing the P2 elements
 -2048: parameter is a vector, its dimension is specified by the
 current literal.
 Other negative values: parameter is a vector, its dimension
 is the value of literal whose MUTL name is P2 negated.

3) TL.PROC.RESULT (INTEGER)

This procedure defines the result type for the procedure (or subroutine).

Parameters:–

P1 Bits 0-13 contain a TYPE SPECIFIER of the result, see
 paragraph 18.3 for its encoding, in addition a zero value
 means the procedure has no result.

4) TL.PROC (INTEGER)

Defines that a procedure (or subroutine) starts at the current position in the code segment. For procedures a name for each of its parameters as specified in the associated procedure specification is automatically allocated at this point.

Parameter:–

P1 MUTL name of procedure.

5) TL.PARAM.NAME (INTEGER, ADDR [LOGICAL8])

This procedure is called during the definition of a procedure to supply symbolic names to its parameters.

Parameters:–

P1 MUTL name of parameter.

P2 Bounded pointer to a byte vector containing the symbolic
 name of the parameter. It is used only for diagnostic purposes.

6) TL.PROC.KIND (INTEGER)

This procedure is called to specify the non-local accessibility of the current procedures variables. Note that TL.PROC.KIND may be called several times before the first imperative statement, in which case only the last call to TL.PROC.KIND is effective.

Parameters:–

P1 bit 1 Specify non-local accessibility of the procedures variables.
 1 – Prohibited
 0 – Allowed.

7) TL.END.PROC ()

Defines the end of the code for the procedure (or subroutine) most recently started.

8) TL.ENTRY (INTEGER)

On calling this procedure the current position in the code is designated to be a multiple entry point for the enclosing procedure. Multiple entry is only allowed on static procedures, all entry points have an identical specification.

Parameters:–

P1 MUTL name previously allocated by calling TL.PROC.SPEC with a P2 value of of %2000, and this name is specified to call the procedure via the multiple entry point.

9) TL.I.PARAM (INTEGER, INTEGER, INTEGER)

MUSS Fortran allows integer data types to be 1, 2 or 4 bytes. In Fortran, procedure specifications are implicit. Therefore the data type precision of integer formal parameters (dummy arguments) cannot be inferred from the actual parameter (argument) when the actual parameter is either a constant or an expression. The procedure TL.I.PARAM allows procedure calls to be made without the precision of the formal parameter being known. Note this procedure is only used for parameters passed by reference and that recursion of such procedures is prohibited.

The procedure allocates MUTL names for two variables. MUTL initialises the first variable to reference the second variable. To generate code to pass parameter P2 of procedure P1 where the precision of the formal parameter is not known, the value is first placed in the second variable, then a reference to it is passed by stacking the first variable as a parameter.

Parameters:–

- | | |
|----|--|
| P1 | MUTL name of procedure. |
| P2 | Number of parameter (counting from zero). |
| P3 | Bit 15. A value of one means that the variables are required for a procedure call.
Bit 14. A value of one means that the variables may be allocated storage in the current data area, which must be static, and the reference between the two variable established.
Bits 0-13 is a TYPE SPECIFIER, see paragraph 18.3 for its encoding. For a procedure call (bit 15 = 1) then the type should be an unbounded pointer where the size specifies the maximum possible precision of the formal parameter. For a procedure definition (bit 14 = 1) then the type specifies the data type of the formal parameter. |

10) TL.BLOCK ()

This procedure is called to indicate the start of a block. No name is allocated for the block.

11) TL.END.BLOCK ()

This procedure is called to indicate the end of the block most recently started.

18.8 LABEL DECLARATION

Labels in code may be classified by their use and their origin, i.e. whether the label is source language or compiler generated. As most compiler generated labels are referenced in a very restricted manner, some MUTL implementations may be able to preserve actual machine registers across such labels. Labels are classified as follows.

- i) Source language labels. Non-local (i.e. from textually embedded procedures) reference to these labels is not permitted. For local jumps to labels of this kind use the $\rightarrow$ orders.
- ii) Source language restart labels. These labels are always accessible non-locally from textually embedded procedures.
- iii) Compiler generated labels.

As some labels need to be forward referenced, a specification of a label is always given before the definition of the value of the label. The specification and declaration of the label must appear in the same procedure.

1) TL.LABEL.SPEC (ADDR [LOGICAL8], INTEGER)

This procedure gives a specification for a label and allocates a MUTL name for the label.

Parameters:—

- | | |
|----|---|
| P1 | Bounded pointer to a byte vector containing the characters of the label's symbolic name. It is used to identify the interface entity and for run time diagnostic purposes. |
| P2 | Bits 0 to 3. Values Interpreted as
0 — Source language label.
> 1 — Compiler generated labels.

Bit 14 = 1 Label to be exported as an interface entity.
Bit 15 = 1 Specification of an assumed interface entity which is to be imported from another module. |

2) TL.LABEL (INTEGER)

This procedure declares a label value by assigning the current code address value to the label.

- | | |
|----|-----------------------|
| P1 | MUTL name of a label. |
|----|-----------------------|

18.9 PICTURE DECLARATIONS

Code planting in MUTL caters for Cobol-like editing operations. The editing is controlled by a picture specification.

1) TL.PIC (ADDR [LOGICAL8])

This procedure allocates a MUTL name for a picture specification.

Parameter:–

P1 This is a bounded pointer to a byte vector containing a picture specification in a canonical form. Exact details of this encoding have not yet been specified.

18.10 CODE PLANTING

Most instructions consist of a function and an operand as specified in the TL.PL procedure. Optimisation of loop control, when possible, is desirable for reasonable object code efficiency. Therefore, MUTL also provides special code planting procedures for the most frequently used types of loops. MUTL distinguishes two special kinds of loops.

- a) Loops that are executed a number of times, where this number can be determined at the start of the loop, and no explicit control variable is required.
- b) Loops that require a control variable, but the increment for the control variable is either +1 or -1.

For (a) the procedures TL.CYCLE and TL.REPEAT are called, for (b) the procedures TL.CV.CYCLE, TL.CV.LIMIT and TL.REPEAT are called. For other loops the appropriate sequence of TL.PL, TL.LABELS, etc., calls must be used.

1) TL.PL (INTEGER, INTEGER)

This procedure specifies one MUTL instruction. It may translate into several actual machine instructions or it may sometimes be optimised out. The latter is very often the case with the D instructions concerned with stepping through data structures. The general form of MUTL instructions

is one-address, and they are in the main orthogonal in the sense that any function can be used with any operand. The exceptions to this rule are given in 18.10.3. A function (FN), given by P1, specifies an operation and an operation type (or MUTL register) as shown in 18.10.1.

The operand part a MUTL instruction (OP), given by P2, is in principle a MUTL name, but there are special encodings of this name to provide for literals, registers, stacked quantities etc. as described in 18.10.2.

18.10.1 Operation Codes

P1 the FN parameter is encoded thus
 bits 0 – 4 specify the operation
 bits 5 – 6 specify the operation set
 0 indicates B operation
 1 indicates A operation
 2 indicates Organisational operation
 3 indicates D operation
 bits 8 - 15 for A and B operations, and ACONV,
 AMODE and ADDR= opcodes this field
 specifies the register number.

There is in effect a greater range of operation types than is apparent in the above since the A-register may be defined to be in any of the following basic modes

REAL(SIZE 32 or 64 or 128 BITS)
 INTEGER(SIZE 8 or 16 or 32 or 64 or 128)
 LOGICAL(SIZE 1 or 8 or 16 or 32 or 64 or 128)
 DECIMAL(any size up to and including 80)
 PTR(SIZE UNBOUNDED or BOUNDED)
 TYPELESS(ANY SIZE or TYPE)

There are other modes for the A-register which cater for specific high level languages. These are known as extended A modes and at present these operations are only available on the A0 register. The extended modes are:

COMPLEX – FORTRAN
 CHARACTER STRING OPERATIONS – FORTRAN and COBOL

Thus in the operation table given below, the A operations are given for each basic mode which is selectable.

			A FUNCTIONS						
	D	B	REAL	DEC	INT	LOG		PTR	ORG.FN
0	=>	=>	=>	=>	=>	=>	=>	=>	STK L
1	=REF	=-	=-		=-			=REF	STK PAR
2	=	=	=	=	=	=	=	=	ENTER
3	SEL FLD	-=			-=	-=			RETURN
4	SEL EL	&			&	&			STK NIL
5	SEL ALT	√			√	√			ACONV
6	BASE	<<		SCALE		<<		BASE	AMODE =
7	LIMIT	>>				>>		LIMIT	STACK
8		+	+	+	+				STK LB
9		-	-	-	-				-> IF =
10		- :	- :	- :	- :				-> IF / =
11		*	*	*	*				-> IF >=
12		/	/	/	/				-> IF <
13		/ :	/ :	/ :	/ :				-> IF =<
14		==			==	==			-> IF >
15		COMP	COMP	COMP	COMP	COMP	COMP	COMP	SEG ->
16									SEG ->
17									AECONV
18			MFN	MFN	MFN				AEMODE =
19		-=>			-=>	-=>			ADDR =
20		& >			& >	& >			
21		√ >			√ >	√ >			BCONV
22									BMODE =
23									
24		+>	+>	+>	+>				
25		->	->	->	->				
26		- :>	- :>	- :>	- :>				
27		*>	*>	*>	*>				
28		/ >	/ >	/ >	/ >				
29		/ :>	/ :>	/ :>	/ :>				
30		==>			==>	==>			
31		**	**		**				

The operators used in the above table have the following meanings:

=> Store.
 =- Load negative.
 = Load.

- =	Logical 'non-equivalence'.
&	Logical 'and'.
V	Logical 'or'.
==	Logical 'equivalence'.
<<	Logical left shift. The operand must be a positive integer.
>>	Logical right shift. The operand must be a positive integer.
+	Add.
-	Minus.
-:	Reverse Minus.
*	Multiply.
**	Exponentiate.
/	Divide. For integer division this yields the nearest integer in the range zero to the mathematical result.
/:	Reverse divide.
COMP	The operand is compared with the register concerned and the T register set accordingly. The T register indicates one or more of the following six states namely, =, /=, >=, >, =<, and <.
-=>	Logical 'non-equivalence' into store.
& >	Logical 'and' into store.
V>	Logical 'or' into store.
==>	Logical 'equivalence' into store.
+>	Add into store.
->	Subtract into store.
-:>	Reverse subtract into store.
* >	Multiply into store.
/ >	Divide into store.
/:>	Reverse divide into store.
SCALE	Scales the decimal A register by a power of 10 specified by bits 0-7 of the operand which are interpreted as a signed integer. For a negative scale factor the digit in bits 8-11 of the operand is added to the last discarded digit in forming the result. Thus a digit of 5 in bits 8-11 rounds and a 0 truncates the result.
STK L	Stack link. Bits 0-11 of the operand contain the name of a procedure specification or zero. The latter value is for implementing interpretive procedure calls. Bits 12 and 13 given information about the procedure, a zero means that there is a definition associated with the specification, a one means entry is via a procedure variable without environment information, and a two means entry via a procedure variable with environment information. This opcode is required at the start of a procedure call sequence.

STK LB	Similar to STK L except the procedure being called is specified by its FIND.N value which is contained in the current literal. The operand is a literal that specifies the type of the result and is encoded as described in paragraph 18.3.
STK PAR	Stacks a parameter in a procedure call sequence. The operand must be a register whose mode is compatible with the parameter specification. For interpretive procedure calls and procedure called with STK LB the mode of the A register is taken as the parameter specification.
STK NIL	In a call of a procedure trailing parameters may be omitted. The operand is a literal that specifies the number of missing parameters.
ENTER	Instruction used at the end of the procedure call sequence to enter the procedure as specified by the associated STK L. A zero operand enters the procedure associated with the specification given by STK L or STK LB, or it specifies the name of a procedure variable, or it specifies the A register in which case the A register contains the entry address of the procedure.
RETURN	Returns control from a called procedure to the instruction immediately following the ENTER used to call the procedure. The operand specifies the result for functional procedures. The result is passed back via the A register. An operand of %30dd means the A register contains the result and a zero operand either means there is no result.
ACONV	Sets a new basic mode and precision for the specified A register, and the contents of A are converted into the new mode as described in 18.10.4. The operand specifies the mode.
AMODE=	Sets the basic mode and precision of the specified A register at the start of an evaluation. The operand is an encoded mode as described in 18.10.4.
BCONV	Sets a new mode and precision for the B register, and the contents of the B register are converted into the new mode as described in 18.10.4. The operand specifies the mode.
BMODE	Sets the mode and precision of the B register.
AECONV	Operates as ACONV but selects an extended mode for the A register.
AEMODE=	Operates as AMODE= but selects an extended mode for the A register.

STACK	Stacks the register specified by the operand. A stacked operand is removed by specifying an UNSTACK operand in the instruction.
>IF=	->IF< ->IF> ->IF/ = ->IF=< ->IF>= Transfers control to a label in the same MUTL segment if the status of T implies the condition of the instruction. Operand is a name of label type.
SEG ->	Transfers control to a label in the same MUTL segment. Operand is of label type.
I SEG ->	Transfers control to a label in another MUTL segment. Operand is of label type.
=REF	Load a reference to the entity specified by the operand. The operand must be the MUTL name of a data item or D[].
SEL FLD	Selects a sub-field of the current field. The operand is a non-negative literal integer.
SEL EL	Selects the Bdd'th element of the current sub-field. The operand is a literal of the form "%20dd where dd specifies a particular B register. A zero operand means select the B0'th element.
SEL ALT	If a field has multiple type definitions this selects the required type definition. A SEL ALT is not required if the first of the multiple type definitions is required. The operand is always a non-negative literal integer.
BASE	The B0 register specifies an element of a vector addressed by a pointer in the A or D register. This pointer is modified by this instruction so that the base of the vector now starts at the element specified. The operand specifies whether to maintain an unbounded or a bounded pointer (i.e. = 0 or 1).
LIMIT	The BO register specifies an element of a vector addressed by a pointer in the D or A register. The pointer is modified so that the last element of the vector becomes the element specified. No operand is required.
MFN	Mathematical and built-in functions that operate on A registers. The operand is taken to be a function number and the available functions are given in 18.10.5 .
ADDR=	Load a MUTL base register. The operand is a literal and it is encoded as follows: Bits 0 – 7 MUTL segment Number or Procedural textual level. Bit 8 1 Load with MUTL segment address 0 Load with stack frame address

Bit 9	0 Plant code to load base register
	1 Note that base register is loaded but do not plant code.

The operation codes for the extended modes are given in the table below:

	COMPLEX	STRING
0	=>	L.STR
1	= -	R.STR
2	=	MOV
3	R=	E.MOV
4	I=	COMP
5		E.COMP
6		SRCH
7		E.SCRH
8	+	LEN
9	-	E.LEN
10	- :	ASC.COMP
11	*	E.ASC.COMP
12	/	LOAD
13	/ :	EDIT
14		C.EDIT
15	COMP	N.EDIT
16		E.N.EDIT
17		MOV.B
18	MFN	
19		
20		
21		
22		
23		
24	+ >	
25	- >	
26	- :>	
27	* >	
28	/ >	
29	/ :>	
30		
31	**	

Where the operation codes mean

R=	Load real part only of complex accumulator
I=	Load imaginary part only of complex accumulator.

The other operators are as for the basic modes except they operate on the complex accumulator.

Nine character string operations are provided:

MOVE	This involves concatenation of one or more strings (called right strings) to form a result string and storing this in a destination string (called a left string). If the result string is longer than the left string, then the string starting at the lefthand end of the result string of the same length as the left string is stored. If the result string is shorter, then the result string is transferred to the lefthand end of the left string and blanks inserted in the remaining part of the left string.
COMPARE	This compares two strings and sets the T status bits accordingly. The strings involved are a left and a right string, and each string consists of a concatenation of one or more strings. The left string is compared with the right string. If one of the string is shorter than the other then it is extended to the right with blanks so that both strings are of the same length. Comparison proceeds by comparing the strings from left to right using the collating sequence of the character set.
SEARCH	This searches one string for the leftmost occurrence of another string. The left string is searched for the leftmost occurrence of the right string. If found its position, counting from one is put in the 8 register, otherwise 8 is set to zero.
LENGTH	This gives the length of a string in B. The left string is used to hold the string.

ASCII COMPARE	Similar to COMPARE but the comparison uses collating sequence described in ANSI, X3.4-1968 (ASCII).
LOAD	This loads a numeric character string, so that an immediately following ACONV opcode enables conversion to a decimal representation in the A register.
EDIT	This performs editing as typified by an alphanumeric move in Cobol between two character strings. Editing is governed by a Picture specifier.
NUMERIC EDIT	This performs editing as typified by a numeric edited move in Cobol between a decimal value in the A register and a character string. Editing is governed by a Picture specifier.
MOVE BYTE	This moves the same literal value to every byte of a character string.

Each one of these operations is specified as a sequence of simpler character string instructions.

Example

E = F//G//H

MUTL instructions

MOV	:: Start of a MOVE operation
L.STR E	:: Left string
R.STR F	:: Result string of right strings F, G and H
R.STR G	:: Concatenated
R.STR H	:: Together
E.MOV	:: End of MOVE operation.

The character string opcodes provided are described below. A string operand is always a vector of byte sized elements.

L.STR	This opcode is used to specify the left string. If the character operation in which this is used permits concatenation of strings, then a sequence of L.STR functions specify that the left strings will be concatenated together and then act as a single string in the operation.
RSTR	As L.STR except it applies to the right string.
MOV	Start of a MOVE operation, this is followed by instructions to specify the left string and then the right string. The left string consists of a single operand, whereas the right string may consist of several concatenated operands.
E.MOV	This opcode terminates the specification of the right string and MOVE operation.
COMP	Start of a COMPARE operation, this is followed by functions to specify the left string and then the right string. Both the left and the right strings may consist of several concatenated operands.
E.COMP	This function terminates the specification of the right string and the COMPARE operation.
SRCH	Start of a SEARCH operation, this is followed by functions to specify the left and then the right string. Both of these strings may consist of several concatenated operands.
E.SRCH	This opcode terminates the specification of the right string and the SEARCH operation.
LEN	Start of a LENGTH operation, this is followed by instructions to specify the right string, which may consist of concatenated operands.

E.LEN	This opcode terminates the specification of the right string and the LENGTH operation.
-------	--

The MOV, E.MOV, COMP, E.COMP, SRCH, E.SRCH, LEN and E.LEN opcodes do not have an operand. The stack may be used during an operation for holding temporary data.

ASC.COMP	As COMP but for ASCII collating sequence.
----------	---

E.ASC.COMP	As E.COMP but for ASCII collating sequence.
------------	---

LOAD	Loads a numeric character string.
------	-----------------------------------

EDIT	Start of an editing operation. The operand is a MUTL name of a Picture specifier. The destination for the edited move is defined by a single left string operand (i.e. L.STR) followed by the source which is defined by a single right string operand (i.e. R.STR).
------	--

E.EDIT	End of an editing operation.
--------	------------------------------

N.EDIT	Start of a numeric editing operation. The operand is a MUTL name of a Picture specifier. The destination of the move is defined by a single left string (i.e. L.STR) followed by code to load the decimal A register.
--------	---

E.N.EDIT	End of numeric editing operation.
----------	-----------------------------------

MOV B	This moves the current literal value to every byte of the character string specified by the operand.
-------	--

18.10.2 Operands

For most functions the operand is one of the following:

- | | |
|---|---|
| 1 | A zero which means the operand is a basic type literal having a value and type of the current basic type literal. |
| 2 | A name of a previously declared item ($1 < OP < 800$). |

- 3 A special operand (OP>%7FF).
- %1002 V.SUB see TL.V.DECL in [18.5](#).
- %1003 The operand is to be popped from the top of the current stack frame. MUTL requires that the use of the stack for temporary variable is determinable at translation time. This means that the corresponding STACK function must statically precede a %1003 operand, and that associated pairs of STACK functions and %1003 operands must be statically nested. As with most other operands of A and B opcodes this operand may be of a different type and size to the register, the operand type information is obtained from the corresponding STACK opcode.
- %1004 D[] the item addressed by the D register.
- %1005 specifies an operand that is normally used with the STACK and the D= instructions. This operand allows the stack to be used for temporary select variables (see [18.5](#)). This operand with a STACK instruction pushes the current selection information of the D register onto the stack, and on the corresponding paired D= instruction with this operand, the stacked selection information is popped into the D register. Obviously several items of this type may be on the stack simultaneously, but as with select variables there are static scope implications. The STACK and D= with this operand may be statically nested, and the D= is paired with the immediately preceding unpaired STACK in the compiled code.
- For the next six special operands the value of the operand for the instruction depends on the state of the T register.
- %1008 a 1 if T implies =, otherwise 0
- %1009 a 1 if T implies /=, otherwise 0
- %I00A a 1 if T implies >=, otherwise 0
- %1008 a 1 if T implies >, otherwise 0
- %100C a 1 if T implies <=, otherwise 0
- %1000 a 1 if T implies <, otherwise 0

%20dd	a B register, bits 0-7 specify which one.
%30dd	an A register, bits 0-7 specify which one.

18.10.3 Restrictions on the use of Operands

There are a number of exceptions to the general rule that any function can be combined with any operand.

- a) Some functions have no operand as stated in the function description
- b) Some functions have specially encoded operands as stated in the function description
- c) Operands of B orders must be of type integer or logical
- d) Operands of store and into store operation codes must match the size and type of the storing register
- e) For MUTL translators where there is only single A and B register the B and A register operands are restricted to =, =>, STACK and STK.PAR opcodes.

18.10.4 Mode Conversions

For most arithmetic functions on the B and A accumulator, if the mode of operand differs from the mode of the accumulator then there is an implicit conversion of the operand to the accumulator mode prior to execution of the function.

The operand of AMODE=, ACONV, BMODE and BCONV operations is taken as an encoded integer where the bits 0-13 give a TYPE SPECIFIER as in [18.3](#) to specify a scalar type mode; an array type generic mode is selected by a value of 2 in bits 0, 1 with bits 2-13 specifying the element type and the number of array elements specified by the current literal. In the case of ACONV bit 14 is also relevant and it specifies a KIND of conversion. Note that basic arithmetic modes have a value less than 256 and have bits 0, 1 zero, while PTR modes have a value of 1 or 3 in bits 0, 1.

For the extended modes the operand of AEMODE= and AECONV operations is encoded as for AMODE= and ACONV, but the TYPE SPECIFIERs are as follows:

Complex – TYPE SPECIFIER of an aggregate type containing 2 reals.
Fortran always uses a TYPE SPECIFIER of %108.

Character string – TYPE SPECIFIER of %83.

When the target type of a conversion is unsigned integer whose precision is the same or greater precision than the existing mode then the conversion is such that a binary equivalent value is obtained. Similarly when converting from an existing mode of unsigned integer to a mode of the same or reduced precision, the conversion is such that a binary equivalent value is obtained. The notion of a binary equivalent is to imply that if a value of one mode is converted to an unsigned integer of greater or equal precision, then a convert back to the original mode yields the same value. For other unsigned conversions then they are similar to integer conversions except that zero extension rather than sign extension occurs.

The conversions available are given below

INTEGER TO REAL

INTEGER TO DECIMAL

REAL TO INTEGER

KIND = 0	Truncated conversion yielding the nearest integer value selected from the range zero to the floating point value.
----------	---

KIND = 1	Rounded conversion yielding the nearest integer value to the the range zero to the floating point value.
----------	--

REAL TO DECIMAL

KIND = 0	Truncated convention.
----------	-----------------------

KIND = 1	Rounded convention.
----------	---------------------

DECIMAL TO INTEGER

DECIMAL TO REAL

INTEGER TO INTEGER Change in precision

REAL TO REAL Change in precision

DECIMAL TO DECIMAL Change in precision

INTEGER TO UNSIGNED INTEGER Sign extend integer to new
precision.

REAL TO UNSIGNED INTEGER

DECIMAL TO UNSIGNED INTEGER

UNSIGNED INTEGER TO UNSIGNED INTEGER
Change in precision.

UNSIGNED INTEGER TO INTEGER Change to new precision.

UNSIGNED INTEGER TO REAL

UNSIGNED INTEGER TO DECIMAL

INTEGER TO PTR The integer value becomes the address ori-
gin of a pointer.

PTR TO INTEGER

KIND = 0 The address origin of the pointer becomes
an integer value.

KIND = 1 The bound of the pointer becomes an inte-
ger value.

PTR TO UNSIGNED INTEGER

UNSIGNED INTEGER TO PTR

INTEGER/REAL TO COMPLEX

KIND = 0 The integer/real accumulator value is converted and becomes the real part, and the imaginary part is set to zero.

KIND = 1 The integer/real value is converted and becomes the real part, and the imaginary part is undefined.

COMPLEX TO INTEGER A truncated conversion is applied to the real part of the complex value to yield an integer value.

COMPLEX TO REAL

KIND = 0 The conversion yields the real part of the complex value.

KIND = 1 The conversion yields the imaginary part of the complex value.

DECIMAL TO NUMERIC CHARACTER STRING

Conversion from decimal to a numeric character string, the result defines a right string operand as does the opcode R.STR. KIND specify the sign representation in the string.

KIND = 0 Trailing sign.

KIND = 1 Leading sign.

KIND = 2 Trailing separate sign.

KIND = 3 Leading separate sign.

NUMERIC CHARACTER STRING TO DECIMAL

Conversion from a numeric character string to decimal, KIND specifies the sign representation as described above.

18.10.5 Mathematical Functions

MFN NO	REAL	INTEGER	DECIMAL	COMPLEX
0	ABS	ABS		ABS
1	MOD.1	MOD.1		CONJG
2	MOD.2	MOD.2		
3	SIGN.1	SIGN.1		
4	SIGN.2	SIGN.2		
5	DIM.1	DIM.1		
6	DIM.2	DIM.2		
7	MAX.S	MAX.S		
8	MAX.M	MAX.M		
9	MAX.F	MAX.F		
10	MIN.S	MIN.S		
11	MIN.M	MIN.M		
12	MIN.F	MIN.F		
13	SQRT	ODD		SQRT
14	SIN			SIN
15	COS			COS
16	LOG			LOG
17	EXP			EXP
18	LOG.10			
19	TAN			
20	ASIN			
21	ACOS			
22	ATAN			
23	ATAN.1			
24	ATAN.2			
25	SINH			
26	COSH			
27	TANH			
28	X SIGN	X SIGN		

Some of the functions (e.g. SIN, ABS) operate solely on the current contents of the accumulator. While others are used in a sequence (e.g. MOD.1,

MOD.2) to provide a ‘function’ that operates on several values, for these the code generation may use the stack for intermediate results, therefore the stack should be in the same state for each function in the sequence.

Example:

For the Fortran statements

```
REAL X,Y,W,Z,V
Z = MAX (X,Y,Z,V)
```

The equivalent MUTL instructions are:

```
A.MODE = real
A = X
MAX.S
A.MODE = real
A = Y
MAX.M
A.MODE = real
A = Z
MAX.M
A.MODE = real
A = V
MAX.F
A => Z
```

In such a sequence the mode of the accumulator must be the same for each of the functions in the sequence.

The functions provided are as follows;

ABS Yields absolute value. For complex mode the result is yielded in the A register whose mode and precision is that of the real part of the complex mode.

MOD.1	Yields $a1 - \text{int}(a1/a2)*a2$,
MOD.2	where $a1$ is the value of the A register for MOD.1, and $a2$ is the value of A for MOD.2.
SIGN.1	Yields $ a1 $ if $a2 \geq 0$. and
SIGN.2	$- a1 $ if $a2 < 0$.
DIM.1	Yields $a1-a2$ if $a1 > a2$. and
DIM.2	0 if $a1 \leq a2$.
MAX.S	Yields the maximum of ($a_s, a_m \dots a_m, a_f$),
MAX.M	where a_s is the value of the A register for
MAX.F	MAX.S, a_m is the value for MAX.M, and a_f for MAX.F.
MIN.S	Yields the minimum of ($a_s, a_m \dots a_m, a_f$)
MIN.M	
MIN.F	
SQRT	Yields square root
SIN	Yields sine
COS	Yields cosine
LOG	Yields natural logarithm
EXP	Yields exponential
LOG10	Yields common logarithm
TAN	Yields tangent
ASIN	Yields arcsine
ACOS	Yields arccosine
ATAN	Yields arctangent

ATAN.1 Yields arctangent of a1/a2
 ATAN.2

SINH Yields hyperbolic sine

COSH Yields hyperbolic cosine

TANH Yields hyperbolic tangent

X SIGN Extract sign. Yields -1 in A register if A value negative, otherwise yields 1 in A register.

ODD Converts the A register into 1 bit mode and loads it with a value of one if it contains an odd integer, otherwise loads a zero.

CONJG Yields the conjugate of the complex value.

The following groups of functions operate on a pair of values, each function is used once and in order.

MOD.1	MOD.2
SIGN.1	SIGN.2
DIM.1	DIM.2
ATAN.1	ATAN.2

The following groups of functions operate on two or more values. The function suffixed S is used for the first value, the one suffixed .F for the last value, and the .M functions for all values inbetween.

MAX.S	MAX.M	MAX.F
MIN.S	MIN.M	MIN.F

Note that intermediate values in the above group functions are held temporarily on the stack, and that nested use of functions is permitted.

2) TL.D.TYPE (INTEGER, ADDR)

The type of an operand referenced by the D register must be set by the compiler whenever a typeless data pointer is loaded into the D register. This procedure is called immediately before the call to TL.PL which loads a

typeless data pointer into the D register. Furthermore this procedure allows the current type information of the operand referenced by the D register to be reset from the parameters.

Parameters:–

- P1 Bits 0–13 contain a TYPE SPECIFIER of the operand addressed by D register, see 18.3 for its encoding.
 Bit 14=0 Resets type information of D when a typeless data pointer is next loaded into D.
 Bit 14=1 Reset type information of D.
- P2 0 – Scalar address by D register
 > 0 – No. of elements in vector addressed by D register or length of bit string.
 < 0 – D addresses a vector but number of elements unknown.

3) TL.CHECK (INTEGER)

This procedure changes the level of program error checking. Note, a change in hardware error detection is dynamic in that it affects all subsequent instructions executed. Whereas the effect of a change to software error detection is static.

At the start of compilation all hardware checking is permitted, and all software checking is inhibited.

Parameters:–

- P1
- | | |
|--------------|--|
| Bit 7 = 0(1) | means change level of software (hardware) checking |
| Bit 8 – 14 | specify which of bits 0-6 are acted upon |
| | If bit (8 + i) = 1) then act on bit i |
| Bit 0 = 0(1) | inhibit (permit) hardware array bounds checking |
| Bit 1 = 0(1) | inhibit (permit) overflow checking on B |
| Bit 2 = 0(1) | inhibit (permit) overflow checking on A in integer mode |
| Bit 3 = 0(1) | inhibit (permit) overflow checking on A in floating point mode |
| Bit 4 = 0(1) | inhibit (permit) overflow checking on A in decimal mode. |

4) TL.RANGE (INTEGER, INTEGER, INTEGER32, INTEGER32)

This plants code to ensure that the value in the register specified by P1 is not outside the value range as specified by P3 and P4.

Parameters:–

- P1 %20dd a B register
 %30dd an A register
 dd specify which register.
- P2 specifies how P3 and P4 are encoded.
 Bit 0 – 0 P3 is an explicit literal
 -1 P3 is the MUTL name of a literal or a variable.
 Bit 1 - 0(1) Upper limit (not) defined.
 Bit 2 - 0 P4 is an explicit literal
 -1 P4 is the MUTL name of a literal or a variable.

5) TL.INSERT (INTEGER)

The binary specified by the parameter is planted in the next available location in the current code area. Thus special operating system functions of an order code, such as dump and reload register, can be used from the systems implementation language

Parameter:–

- P1 Value of binary to be planted. The unit of binary planted is machine dependent.

6) TL.CYCLE (INTEGER)

This indicates the start of a loop which will be executed P1 times.

Parameter:–

- P1 The MUTL name of an integer variable or an integer literal, note that for integer variables MUTL uses the variable specified in the limit test for the loop and does not take a copy of the variable. A value of zero means use the current literal. In addition the name %30dd (see TL.PL) may be used.

7) TL.REPEAT ()

This defines the end of a loop.

8) TL.CV.CYCLE (INTEGER, INTEGER, INTEGER)

This indicates the start of a loop which has an explicit control variable and a unit increment.

Parameters:—

P1	The MUTL name of the integer control variable.
P2	The initial value to be assigned to the control variable at the start of the loop. It can be the MUTL name of a variable or a literal, or zero meaning use the current literal, or %30dd or %1004 (see TL.PL).
P3	
Bit 0,1,3 0	Increment is +1 and the loop is terminated when the control variable is greater or equal to the limit specified in TL.CV.LIMIT.
1	As 0 except loop is terminated on a greater than condition.
2	Increment is -1 and the loop is terminated when the control variable is less than the limit specified in TL.CV.LIMIT.
3	As 2 except loop is terminated on a less than or equal to condition.
4	(i.e. bit 3 = 1) Increment is +1 and loop is terminated when the control variable is equal to the limit specified in TL.CV.LIMIT.
6	As 4 but increment is -1.
Bit 2=0	Control variable may be left undefined at the end of the loop.
Bit 2=1	Control variable is defined at the end of the loop.

9) TL.CV.LIMIT (INTEGER)

This is called after TL.CV.CYCLE, but before the start of the code for the loop body. This procedure specifies the limit value of the termination test of the loop.

Parameter:–

P1 This can be the MUTL name of an integer variable of the same size as the control variable or an integer literal, or zero meaning use the current literal, or %30dd (see TL.PL).

10) TL.REG (INTEGER)

Once a register is loaded it is defined to be in current use. A register is no longer in current use after one of the following events:

<u>EVENT</u>	<u>REGISTER</u>
a) => and into store orders	B, A or D
b) COMP order	B, A or D
c) ST PAR with register operand	B,A or D
d) STACK with register operand	B,A or D
e) B or A used as a register operand when function is not =>	B or A
f) access of operand via D[]	D
g) SEL.EL	B
h) after a TL.MAKE function call	B
h) IF orders	T
i) BASE and LIMIT orders	B
j) TL.CYCLE, TL.CV.CYCLE, TL.CV.LIMIT with %30dd operand %A	

or events a) to g) inclusive, the A and B registers may be retained in current use after the event by calling TL.REC immediately before the TL.PL call which causes such an event. The D register is only maintainable in current

use after event a) when the operand is not a select variable and after f).

In other situations the compiler may need to inform the target code translator that a register is no longer in use.

Parameters:–

P1 bit – 0 = 1 B }
 1 = 1 A } retain register in current use
 2 = 1 D }
 3 = 1 T }

 bit – 4 = 1 B }
 5 = 1 A } register no longer in current use
 7 = 1 ADDR }

 bits 8 – 15 register number.

18.11 MUTL INITIALISATION

1) TL (INTEGER, ADDR [LOGICAL8], INTEGER)

Initialises MUTL for a compilation. This initialisation includes creating MUTL segment 0, this may be overridden where necessary by a call to TL.SEG to replace this default creation of MUTL segment 0. At the start of each module, area 1 of the module is automatically mapped to this code segment, and area 1 selected as the current code segment while area 0 (i.e., the stack) is selected as the current data area. The action of MUTL in this mode can be described more precisely in terms of MUTL procedure calls.

1) At TL time, MUTL takes the following action:

a) TL.SEG (0, Default segment Size. -1, -1, 6).

2) At TL.MODULE time. MUTL takes the following action:–

- a) Map area 1 (code area) of the module to segment 0: TL.LOAD(0,1).
- b) Select area 1 as the current code area: TL.CODE.AREA(1).
- c) Select area 0 as the current data area: TL.DATA.AREA(0).

- 3) At TL.END time, MUTL plants in code segment 0 any code required for exiting from a program or library.

There are some restrictions when compiling in a one pass mode. In order to map data declarations as they occur interface literals and anonymous types must be exported before they are imported into any other modules. Furthermore on some machines there may be some restrictions on the use of interface data variables.

When compiling a library P3 specifies the maximum number of procedures expected in the library interface.

Parameters:-

P1 Bit 0 = 0	Produce necessary data structures to support run time diagnostics
Bit 0 = 1	Do not produce data structures
Bit 1 = 0	Normal compilation run
Bit 1 = 1	MUSS type compilation, note that this means that all on-stack declaration are within procedures.
Bit 2	Only used for a normal compilation run. A value of zero means compilation of all or part of a program, and a value of one means compiling all or part of a library.
Bit 3	This bit when set indicates that calls to library procedures will only be compiled for libraries which were opened for cross-compilation (i.e. %20 bit in mode parameter of LIB).
Bit 4	A zero means that a faulty compilation yields no output files, and a one means output files are produced.
Bit 5	The use of this bit is machine dependent. For example when compiling MUSS control of the stack position may be required. See the MUTL implementation manual of the appropriate machine for details.
Bits 6-7	This specifies the form of the MUTL output as follows. 0 -Generate executable binary. 1 -Generate MUBL. 2 -Generate no code.

P2	This gives the name of the file into which the output (i.e. code and data areas etc.) of a successful translation will be placed at the end of a translation (i.e. at TL.END).
P3	
Bits 0-11	Specify the number of procedures in library interface.
Bit 15	A value of one means that MUTL will not create a library entry table.
Bits 12-14	Specify library organisation dependent information. See appropriate Library Interface Implementation Manual for details.

2) TL.END ()

Terminates a compilation.

3) TL.MODULE ()

Initialises the compilation of a module.

4) TL.END.MODULE (INTEGER)

Terminates the compilation of a module.

Parameter:—

P1	A zero value means no faults detected by high level language compiler in this module, a value of one means faults.
----	--

5) TL.GLOBAL ()

TL.GLOBAL is called immediately prior to declaring a global entity (see [18.2.4](#)) to MUTL.

6) TL.MODE (INTEGER, INTEGER)

This procedure allows the translation mode to be altered. The mode is set first by P1.

- P1 Bits 8-15 specify which bits of the translation mode to alter. If bit 8 is set then change bit 0, if bit 9 is set then change bit 1, etc. Bits 0-7 specify the altered bit values of the translation mode. E.g. a mode of %8480 would change bit 3 to a zero and bit 7 to a one. The translation mode is interpreted as follows:
 Bit 0=1 Inhibit repositioning of fields in a type
 Bit 1=1 Inhibit code optimisation.
 Bit 2=0 Align fields of a type to minimise store accessing.
 Bit 2=1 Inhibit above alignment thus reducing the size of some types.
 Bit 3=1 Relax type checking of interface variables of user defined type so that only the size of types are checked.
 Bits 4-7 have machine dependent encodings The MC68000 encodings are
 Bit 4=0 Compile MC68010 instructions.
 Bit 4=1 Compile MC68000 instructions.
- P2 This parameter permits further machine dependent control over the translation. See the implementation manual of the appropriate machine for details.

18.12 TARGET MACHINE PARAMETERS

It was mentioned earlier that in the interest of efficiency and reducing the complexity of a MUTL translator on optimising compiler is aware of certain parameters concerning the target machine being compiled for.

1) TL.ENQ (INTEGER) INTEGER

This procedure yields information about the compiler target machine. This information is returned as the result of the procedure.

Parameters:

- P1 = 0 return the MUTL type encoding for default length integers of the machine.
 P1 = 1 return the MUTL type encoding for default length reals of the machine.
 P1 = 2 return the number of procedural textual levels maintained by a display.

2) TL.ENQ.REG (INTEGER)ADDR [INTEGER]

This procedure yields information concerning the number of MUTL registers available on the target machine. For a particular high level language MUTL allocates a fixed number of B registers and a fixed number of Base registers, but the mapping of A registers is not so straightforward. On many machines there are separate registers for different arithmetic modes, also a large register is often comprised of two or more smaller registers.

The result of the procedure specifies the available registers, and is a pointer to a vector of integers, encoded as follows:

Word 0	Number of B registers
Word 1	Number of Base registers
Word 2	0
Word 3	Number of A register entries. Each entry consists of the three following words:
Word 0	Specifies which arithmetic modes that this set of machine registers support Bit 0 – 1 Real Bit 1 – 1 Integer/Logical
Word 1	Machine register size specified in bytes. If a register greater than this size is required then two or more consecutive registers are allocated as a single MUTL register. For example if a machine has four 32 bit registers, then the first two or the last two could be treated as a 64 bit register. Note that when several registers are needed, the number of registers is always a power of two (2^n).
Word 2	Number of machine registers in this set.

The MUTL register AO operates in all modes.

Thus an example result vector encoding is

2	:: 2 MUTL. B registers i.e. B0, B1
3	:: 3 MUTL Base registers i.e. BASE0, BASE1, BASE2
2	:: Machine has 2 kinds of arithmetic registers.

0	
1	:: Set of Floating point registers.
8	:: 64 bit precision.
7	:: Seven registers of this kind numbered :: A1 to A7.
2	:: Set of 4 integer registers.
4	:: 32 bit precision.
4	:: Four registers numbered A8 to A11.

When several machine registers are treated as a single MUTL register the MUTL register number is that of the first register allocated, e.g. if A10 and A11 are treated as a 64 bit register, then A10 refers to the register pair. Furthermore compilers must choose consecutive registers such that they start on a register boundary within the register set which is an exact multiple of the number of machine registers needed for a particular A register. Thus A10 is a valid 64 register but A9 is not.

Parameters:–

MODE Some features of MUTL necessitate registers for their implementation, thus if such registers are not needed in a particular high level language, such registers will, wherever possible, be made available as additional MUTL registers. The MODE parameter indicates this language dependence as follows:

- 0 – MUSL
- 1 – FORTRAN
- 2 – PASCAL

result A bounded pointer to a vector of integers containing the Register Resource and Register Mapping tables.

18.13 DIAGNOSTICS

Run time diagnostics refer to items in terms of the source language program. Most of the necessary information to accomplish this is in the existing declarative procedures within MUTL, e.g. TS.DECL, TL.TYPE etc. Alternatively, or in addition to high level language diagnostics, a compile and data

map of the compilation may be produced from the diagnostic information known at compile time. The procedures in this section exist solely to supply information to support diagnostics.

1) TL.LINE (INTEGER32) INTEGER

MUTL notes the specified source language line number. This procedure should be called for each line of source program that generates code.

TL.LINE yields the address of the next location in the code area to enable compilers to incorporate this information in a compile map.

Parameters:—

P1 = 0	means that the line number is obtained by the basic system IN.LINE procedure.
≠ 0	means that the parameter itself specifies the line number.

2) TL.SOURCE (ADDR [LOGICAL8])

Many program development tools have a hierarchical source text representation of a program. For example, Floccoder or structure charts. TL.SOURCE informs MUTL of the correspondence between this external source text and the object program. The source text is considered as a hierarchy of named source blocks. The parameter to TL.SOURCE specifies the level and name of each source block. TL.SOURCE is also called to terminate each source block.

The first character of the P1 string is interpreted as follows:

A – Y	Start of source blocks at level 1 (A) to 25 (Y). The remaining characters specify the name of the source block.
Z	Terminates a source block and starts a new source block at the same source block level. The remaining characters specify the name of the new source block.

A nil string for P1 terminates a source block. A P1 string of ‘|’s terminates two or more source blocks. The number of ‘|’s plus one indicates how many source blocks to terminate.

3) TL.CUR.SOURCE ()

This procedure prints the current source position on the current output stream. This procedure enables compilers to give the source language position for compile errors.

4) TL.BOUNDS (INTEGER, ADDR (INTEGER))

A vector variable in MUTL consists of a single dimension with a lower bound of zero. Languages that support multi-dimensional arrays, non-zero bounds or adjustable dimensions may elect (for reasons of efficiency) to map these arrays into a signal dimension with a zero bound. TL.BOUNDS function is therefore to define this mapping, so that array diagnostics are in terms of the source language declaration of the array.

A multi-dimension array M may be mapped to a MUTL vector V in two ways. Consider an array of N dimensions with lower and upper bounds on each dimension of L_i, U_i for $i = 1, 2 \dots N$, and an array element $M(S_1, S_2 \dots S_n)$. A 'Forward Mapping' is such that $M(S_1, S_2 \dots S_n)$ is mapped to

$$V((S_1 - L_1)*D_2*D_3 \dots *D_n + (S_2 - L_2)*D_3*D_4 \dots D_n + (S_n - 1 - L_{n-1})*D_n + (S_n - L_n))$$

A 'Reverse Mapping' is such that $M(S_1, S_2 \dots S_n)$ is mapped to

$$V(S_1 - L_1 + (S_2 - L_2)*D_1 + (S_3 - L_3)*D_2*D_1 \dots (S_n - L_n)*D_{n-1}*D_n \dots *D_1)$$

where D_i is the size of the i th dimension i.e. $(U_i - L_i + 1)$.

Parameters:—

P1 Bits 0 - 11 specify the MUTL name. A value of zero in bit 13 means the array has a Forward Mapping and value of one means it has a Reverse Mapping.

P2 Bounded pointer to vector of integer elements. The bounds of a dimension are specified by a pair of elements. The first element specifies the lower bound and the second the upper bound of the dimension. Each element contains a MUTL name of a literal or a scalar variable. However a value of zero is permitted for U1 of a Forward mapped array, and for Un of a Reversed mapped array; this means that the bound is not known.

5) TL.PRINT (INTEGER)

 This procedure controls the monitoring output of MUTL.

Parameter:—

P1 This is bit encoded.

 Bit 0 = 1 means print name of all TL procedure calls and their parameters.

 Bit 2 = 1 means print procedure and segment map.

 Bit 3 = 1 means print code generated.

 Bit 4 = 1 means print data map.

 Bit 6, 7 used by implementers of MUTL for debug monitoring control. Object code in MUTL is produced from an Intermediate form held in a vector. Setting bit 7 = 1 prints out this vector prior to code generation. See Section 27 of the MUTL Implementation Description for further details of the encoding of this intermediate form. Bit 8 = 1 means insert tracing instruction at the beginning of each line.

MUSS

USER MANUAL

VOLUME 1

CHAPTER 19

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED JUNE 1986

19 COMPILER WRITER UTILITIES

This Chapter describes utilities to aid compiler construction and development.

19.1 SYNTAB

SYNTAB is a Syntax Processing Package which allows a translator written largely in MUSL to have its syntax phase defined by a notation similar to that of BNF. First the package translates the syntax specification into MUSL then sends the resulting code to a file for compilation by the MUSL compiler. This MUSL code needs a SCAN procedure which may be called within the compiler. When entered this procedure attempts to match some text supplied in a vector parameter with the automatically generated vector containing the encoded form of the specified syntax. The SCAN procedure uses a top-down fastback technique.

19.1.1 The Notation for describing syntax

The syntax specifications must appear at the beginning of the translator and be delimited by:—

```
BEGIN SYNTAX SPEC
and END
```

Thus a translator has the form:—

```
BEGIN SYNTAX SPEC
<SYNTAX SPEC>
END
<AUTOCODE PROGRAM>
```

There are five components in the SYNTAX SPEC the main two of which define the synes and cosynes.

```

<SYNTAX SPEC> =
<DELIMITER LIST>
<PROCEDURE LIST>
<COSYNE LIST>
<SYNE DEFINITIONS>
<COSYNE DEFINITIONS>

```

Syne definitions are syntactic rules of the languages written as in BNF except for these differences:—

1. Stylistic differences. The word SYNE must precede each definition and ::= is replaced by =.
2. Differences in ordering. The order of alternatives and of elements within alternatives are both different. Syne formulae are used by a left to right scanning algorithm which requires that:—
 - (a) Any alternative which is a stem of another comes after it.
 - (b) If one alternative is a special case of another it must come first.
 - (c) In recursive definitions there must be at least one leftmost element not recursive.
3. Metalinguistic Bracketing. Several alternatives may be specified as an element of another by enclosing them in square brackets.
4. Those syne definitions which are to be referenced by the autocode part of the compiler, (e.g., as parameters of the scan routine) must be preceded by a *. The result of this will be that a CONST declaration is generated to associate the syne name with the position allotted to it in a DATA VEC 'SYNES'.
5. Alternatives within syne definitions must be less than 128 elements long.

For example, a definition of <STATEMENTS> might take the form:

```
SYNE <STATEMENTS> = <STATEMENT> [<STATEMENTS> | <NULL>]
```

where <NULL> indicates an empty string.

For both semantic and syntactic reasons it is convenient to be able to interrupt the scanning algorithm at defined points and execute code provided by the user. This is achieved by inserting an element into the syntax which is referred to as if it were a syne but is in fact defined as the name of a section of code (COSYNE) to be executed when the scanning algorithm reaches that point.

To make this facility more flexible the cosyne routine may have one numeric parameter, the value of which is defined explicitly wherever the cosyne is used thus:

<cosyne name (numeric parameter)>

Cosynes are defined as labelled sequences of Autocode instructions which operate within the SCAN procedure. Each sequence is labelled with the name of the cosyne it represents. Cosynes may use the variables and labels in the scan of which the following are the most useful:—

AS, WS, LINE, SYNTAX

Descriptors mapping the analysis record area, the working stack, the current statement and the syntax tables, respectively.

AP, WP, SS, SY

Pointers to the current position in AS, WS, LINE and SYNTAX, respectively

TRUE, FALSE

Labels to which the cosynes may jump on completion.

In a translator it is usual to itemise the input string before applying the scan. This pre-scan pass may reduce delimiters to single pseudo symbols. A similar transformation has to be applied to delimiters appearing in the syntax. These delimiters and the value of the code to be assigned to them must be listed thus:—

```
< DELIMITER LIST> =
                        DELIMITERS<NL>
                        <DELIMITER SPEC>
                        END<NL>
```

where

```
<DELIMITER SPEC> = <INTEGER>/<STRING><NL>[<DELIMITER
SPEC> | <NIL>]
```

<INTEGER> is any integer in the range 0 – 255 and is the code to be assigned to the delimiter specified by,

<STRING> which is any character string but all strings must begin with the same symbol, (e.g., for delimiters in quotes),

<NL> represents a newline.

Each procedure name used as the parameter of a cosyne will be replaced by the integer which indexes its entry in the procedure list

Formally:

<PROCEDURE LIST> = PROCEDURE NAMES<NAME LIST><NL>

where <NAME LIST> = <NAME>[,<NAME LIST> | <NIL>]

The COSYNE LIST contains the names of all the cosyne used in the syntax. Its form is:

COSYNE NAMES<NAME LIST><NL>

FIGURE 1
Schematic Form of Input

```
BEGIN SYNTAX SPEC
DELIMITERS
128/'BEGIN'
.
.
END

PROCEDURE NAMES name, name, .....
COSYNE NAMES name, name, .....
SYNE<name>= - - -
*SYNE<name>= - - -
.
.
COSYNES
.
.
END SYNTAX SPEC
```

FIGURE 2
Schematic Form of Output
for a MUSL Compiler

```

LITERAL synename=index;
.
.
DATAVEC SYNTAX($L08)
.
.
END;
PSPEC SCAN(ADDR[$L08],ADDR[$IN],ADDR[$1N],ADDR[$IN],$IN);
PROCEDURE SCAN(SYNTAX,LINE,AS,WS,START);

  - >PASSWITCH;
  LOOP2:
  SWITCH SYNTAX[SY]\
  cosyne name.
  .
  .
  cosyne name;
  .
  PASSWITCH;;
  .
  **IN -1

```

19.1.2 The Syntactic Scan Procedure

The scan procedure has the specification:–

```

PROC SPEC SCAN (ADDR [LOGICAL8], ADDR [LOGICAL8],
                ADDR [LOCICAL8], ADDR [LOGICAL8], INTEGER32)

```

The parameters are:–

- P1 The string SYNTAX which contains the complete syntax to be used in the scan and which is set up by the syntax processing phase. The scan maintains a modifier SY (I32 variable) at the current position in the syntax tables.

- P2 The vector LINE with 8-bit or 32-bit elements containing the text to be recognised. Each element is compared with a byte in the syntax tables. SS is maintained pointing to the last symbol recognised. On entry SS = 0.
- P3 and P4 Two vectors AS and WS with 6-bit or 32-bit elements as the user requires. The modifiers within these vectors are AP and WP respectively, zeroed on entry to the scan. AS and WS are for use by the cosynes in generating an analysis record. AS or analysis space is conventionally the area in which the final analysis record is built up. WS or work space is the area used as temporary storage for intermediate levels of the analysis record. These areas are used for this purpose by the built-in cosynes VAL, NOVAL and IN. WS is used for output information as follows:—
- WS[0] = -1 if the scan fails, WS[0] = 1 if it passes.
- Consistent use of VAL will ensure that WS[1] is a modifier in AS pointing to the first element of the analysis record. The first free space in WS, usually WS[2] is the value of the modifier in LINE (SS) for the last symbol recognised. The analysis record created by the user starts as WS[1]. AS[0] contains the number of elements in the complete analysis record (i.e., if the last element is AS[4], AS[0] = 5).
- P5 The address of the start of the SYNE to be scanned for. This is the INTEGER32 literal constant given by the name of a starred SYNE.

19.1.3 Operation of the Scan Procedure

19.1.3.1 Analysis Records

Obviously in BNF no record is kept of the way in which the source string fits the syne definitions. Also there is no specific way of recognizing the end of an alternative. Both these functions are performed by cosynes. A terminal cosyne is required at the end of every alternative in a syne definition. This should generate the required record and end by transferring control to a specified point in the scanning routine, where the stack will be adjusted and the scan continued in the syne definition one level up. Two terminal cosynes

are built into the system, they may be used as models for others. The first is UP which causes the scan to continue at the syne definition one level up and generates no analysis record. The other, VAL, generates an analysis record which has a tree structure form. The parameter of the reference to the VAL cosyne is entered as the first element of the current level of tree structure.

On some occasions it may be necessary to record information in the analysis record without creating a new level of tree structure. This is achieved by use of the cosyne IN, which inserts its parameter into the analysis record. Thus analysis records may be produced as in Figure 3 19.1.3.1. The pointers are stored as indices in the vector containing the analysis record.

```

SYNTAX:
  SYNE <SENTENCE> = THE[CAT<IN(1)>|DOG<IN(2)>]
                    <VERB>THE WALL<VAL(0)>
  SYNE <VERB>      = SAT<PREP><VAL(1)>|RAN FROM <VAL(0)>
  SYNE <PREP>      = [ON<IN(0)>|UPON<IN(1)>]<UP>
INPUT:
      THE DOG SAT ON THE WALL
ANALYSIS RECORD:
      |
      |
      |-----|
      |0|2|. |
      |-----|
      |
      |
      |-----|
      |1|0|
      |-----|

```

Figure 3. Analysis Records

19.1.3.2 Cosynes

These are labelled sections of code usually delimited by BEGIN and END. Several are built into the scan procedures and these are:-

BRANCH, SYMBOL, SYNE, MERGE, UP,
ENDFALSE, ENDTRUE, VAL, NOVAL, IN

The VAL, UP and IN cosynes have already been discussed. The cosyne

NOVAL acts as VAL but has no parameter and does not precede the analysis record level with a number. The other built in cosynes are of less interest to the user.

A reference to the UP cosyne occupies one 8-bit element of the Internal encoding of the syntax; a syne reference occupies 3 elements and all other cosyne references occupy 2 elements.

The second of the 2 elements occupied by a cosyne holds the parameter and this may be accessed within the cosyne as:–

SYNTAX1[SY]

Cosynes may insert information onto the current level of analysis record by storing to:–

WS[WP]

and advancing the index WP. They may also access the next symbol to be recognised as:

LINE[SS + 1]

If a symbol is recognised thus, then the index SS should be advanced.

Cosynes should normally exit to one of the two built in labels TRUE or FALSE – according to whether recognition is to proceed or to backtrack in the syntax.

MUSS

USER MANUAL

VOLUME 1

APPENDIX 1

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED APRIL 1986

A1 APPENDIX 1 – CHARACTER CODES

A standard character set, based on the ISO representation, is used throughout MUSS.

Hex Code	Hex Code	Hex Code	Hex Code
00 Null	20 space	40 @	60 ‘
01 SOH (start of heading)	21 !	41 A	61 a
02 STX (start of text)	22 "	42 B	62 b
03 ETX (end of text)	23 #	43 C	63 c
04 EOT (end of transmission)	24 \$	44 D	64 d
05 ENQ (enquiry)	25 %	45 E	65 e
06 ACK (acknowledge)	26 &	46 F	66 f
07 BEL (bell)	27 '	47 G	67 g
08 BS (backspace)	28 (	48 H	68 h
09 HT (horizontal tab)	29)	49 I	69 i
0A LF (line feed)	2A *	4A J	6A j
0B VT (vertical tab)	2B +	4B K	6B k
0C FF (form feed)	2C '	4C L	6C l
0D CR (carriage return)	2D -	4D M	6D m
0E SO (shift out)	2E .	4E N	6E n
0F SI (shift in)	2F /	5F O	6F o
10 DLE (data link escape)	30 0	50 P	70 p
11 DC1 (XON)	31 1	51 Q	71 q
12 DC2	32 2	52 R	72 r
13 DC3 (XOFF)	33 3	53 S	73 s
14 DC4	34 4	54 T	74 t
15 NAK (negative acknowledge)	35 5	55 U	75 u
16 SYN (synchronous idle)	36 6	56 V	76 v
17 ETB (end of trans. block)	37 7	57 W	77 w
18 CAN (cancel)	38 8	58 X	78 x
19 EM (end of medium)	39 9	59 Y	79 y
1A SUB (substitute)	3A :	5A Z	7A z
1B ESC (escape)	3B ;	5B [	7B {
1C FS (file separator)	3C <	5C &	7C
1D GS (group separator)	3D =	5D]	7D }
1E RS (record separator)	3E >	5E ↑	7E ~
1F US (unit separator)	3F ?	5F -	7F del

Some devices on some MUSS systems have soft fonts. In this case there are some built-in fonts which can be used from TXT facility.

<u>Hex Code</u>	<u>font 1</u>	<u>font 2</u>	<u>font 3</u>	<u>font 4</u>	<u>font 5</u>	<u>font 6</u>
20	space	space	space	space	space	space
21	!	!	!	!	!	!
22	"	"	"	"	"	"
23	#	£	£	£	£	■
24	\$	\$	\$	\$	\$	∫
25	%	%	%	%	%	%
26	&	&	&	&	&	&
27	,	,	,	,	,	,
28	(	(	(	(	(	(
29	)	)	)	)	)	)
2A	*	*	*	*	*	*
2B	+	+	+	+	+	+
2C	,	,	,	,	,	,
2D	–	–	–	–	–	–
2E	.	.	.	.	.	.
2F	/	/	/	/	/	/
30	0	0	0	0	0	0
31	1	1	1	1	1	1
32	2	2	2	2	2	2
33	3	3	3	3	3	3
34	4	4	4	4	4	4
35	5	5	5	5	5	5
36	6	6	6	6	6	6
37	7	7	7	7	7	7
38	8	8	8	8	8	8
39	9	9	9	9	9	9
3A	:	:	:	:	:	:
3B	;	;	;	;	;	;
3C	<	<	<	<	<	<
3D	=	=	=	=	=	=
3E	>	>	>	>	>	>
3F	?	?	?	?	?	?
40	@	@	@	@	@	–

<u>Hex Code</u>	<u>font 1</u>	<u>font 2</u>	<u>font 3</u>	<u>font 4</u>	<u>font 5</u>	<u>font 6</u>
41	A	A	A	<i>A</i>	A	A
42	B	B	B	<i>B</i>	B	B
43	C	C	C	<i>C</i>	C	χ
44	D	D	D	<i>D</i>	D	Δ
45	E	E	E	<i>E</i>	E	E
46	F	F	F	<i>F</i>	F	Φ
47	G	G	G	<i>G</i>	G	Γ
48	H	H	H	<i>H</i>	H	H
49	I	I	I	<i>I</i>	I	I
4A	J	J	J	<i>J</i>	J	
4B	K	K	K	<i>K</i>	K	K
4C	L	L	L	<i>L</i>	L	$\wedge$
4D	M	M	M	<i>M</i>	M	M
4E	N	N	N	<i>N</i>	N	N
4F	O	O	O	<i>O</i>	O	O
50	P	P	P	<i>P</i>	P	P
51	Q	Q	Q	<i>Q</i>	Q	Θ
52	R	R	R	<i>R</i>	R	ρ
53	S	S	S	<i>S</i>	S	Σ
54	T	T	T	<i>T</i>	T	τ
55	U	U	U	<i>U</i>	U	Υ
56	V	V	V	<i>V</i>	V	Σ
57	W	W	W	<i>W</i>	W	Ω
58	X	X	X	<i>X</i>	X	Ξ
59	Y	Y	Y	<i>Y</i>	Y	Ψ
5A	Z	Z	Z	<i>Z</i>	Z	Z
5B	[	[	[	[	[	[
5C	\	\	\	\	\	\
5D	[	[	[	[	[	[
5E	↑	↑	↑	↑	↑	↑
5F	-	-	-	-	-	-
60	`	`	`	`	`	`

Hex Code	font 1	font 2	font 3	font 4	font 5	font 6
61	a	a	a	<i>a</i>	a	α
62	b	b	b	<i>b</i>	b	β
63	c	c	c	<i>c</i>	c	γ
64	d	d	d	<i>d</i>	d	δ
65	e	e	e	<i>e</i>	e	ϵ
66	f	f	f	<i>f</i>	f	ε
67	g	g	g	<i>g</i>	g	ζ
68	h	h	h	<i>h</i>	h	η
69	i	i	i	<i>i</i>	i	θ
6A	j	j	j	<i>j</i>	j	ϑ
6B	k	k	k	<i>k</i>	k	κ
6C	l	l	l	<i>l</i>	l	λ
6D	m	m	m	<i>m</i>	m	μ
6E	n	n	n	<i>n</i>	n	ν
6F	o	o	o	<i>o</i>	o	ξ
70	p	p	p	<i>p</i>	p	π
71	q	q	q	<i>q</i>	q	ρ
72	r	r	r	<i>r</i>	r	ρ
73	s	s	s	<i>s</i>	s	v
74	t	t	t	<i>t</i>	t	v
75	u	u	u	<i>u</i>	u	v
76	v	v	v	<i>v</i>	v	ν
77	w	w	w	<i>w</i>	w	ω
78	x	x	x	<i>x</i>	x	x
79	y	y	y	<i>y</i>	y	y
7A	z	z	z	<i>z</i>	z	z
7B	{	{	{	{	{	{
7C						
7D	}	}	}	}	}	}
7E	~	~	~	~	~	~
7F	delete	delete	delete	<i>delete</i>	delete	delete

MUSS

USER MANUAL

VOLUME 1

APPENDIX 2

UNIVERSITY OF MANCHESTER
INSTITUTE OF SCIENCE AND TECHNOLOGY

ISSUED APRIL 1986

A2 APPENDIX 2 – FAULT REASONS

Class 0 – Program errors

These fault reasons are bit significant and so multiple error conditions may be signalled.

Code

%1	Arithmetic trap
%2	Illegal instruction/operand
%4	Disc transfer fails
%8	Access violation
%10	Segment undefined
%20	Segment overflow
%40	Stack overflow
%80	Illegal organisational command call
%100	Break point type 1
%200	Break point type 2
%400	Program counter set to odd address
%800	Bound check fail
%1000	Store limit exceeded

Class 1 – Limit violations

Code

1	CPU runout
---	------------

Class 2 – Timer runout

Class 3 – External interrupts

The reason for an external interrupt is bit significant and so multiple interrupt conditions may be signalled.

Code

%1 to %80	Break interrupt from the devices connected to channels 0 to 7 respectively
%100 to %8000	Message interrupt on channels 0 to 7 respectively

Class 4 — Organisational command errors

Code

- 1 Segment already exists
- 2 No segments available in virtual store
- 3 Illegal segment number
- 4 Illegal size requested
- 5 Illegal access permission requested
- 6 Segment does not exist
- 7 System unable to create segment (no resources)
- 8 Validate fail
- 9 Unable to map segment
- 11 Illegal channel number
- 13 No messages on specified channel
- 14 System unable to send message (no resources)
- 15 Channel closed
- 16 Faulty message sent to file manager
- 17 No response from destination process after send message and suspend.
- 20 System process number or process identifier invalid
- 21 Current process not supervisor
- 23 System unable to create a process (no resources)
- 25 Illegal username or password
- 30 Cannot create enough space for screen editor
- 32 Process name already exists
- 33 Process does not exist
- 35 No free tasks
- 36 Invalid task number
- 41 File store limit exceeded
- 42 Subdirectory does not exist
- 43 File name does not exist
- 44 Exclusive access deadlock
- 45 Unauthorised access requested
- 46 Illegal segment specified
- 47 File name already exists
- 49 Disc transfer failed
- 50 Unable to update common segment
- 52 Disc unavailable
- 56 System unable to create a remote directory (no resources)
- 60 Semaphore already defined
- 61 System unable to create semaphore

62	Semaphore not defined
64	Invalid semaphore operation
70	Mount abandoned
71	Label incorrect
72	Buffer non-existent or too small
73	Illegal unit number
74	EOT/TM encountered
75	MT operations failed
76	Unit not assigned to process
79	Invalid disc address
80	Subords limit reached
81	User name not unique
82	No more spare user available
83	User has used reusables
84	User name expected
85	Resource not available
86	Not authorised to payout
87	Parameter out of range
86	Current user not superior
69	Cannot erase default user
90	Device out of range
91	Invalid configuration parameter
93	Invalid peripheral channel
94	Device not connected to process

Class 4 — errors are initially returned to a process by a negative
type code in PW0

Class 5 — Input/Output set up errors

Code

1	No streams available
2	Current File not available
3	Illegal stream name (in STRn*)
4	Stream undefined
5	Incorrect type of stream
6	Incorrect type of document
7	Unable to open file
8	Unable to connect to device

Class 6 Input/Output errors

Code

- 1 Input ended
- 2 Attempt to read/write beyond end of unit or record
- 3 Output limit exceeded
- 4 Unable to create segment for buffered output document
- 5 Unable to file/send output document
- 6 Destination process does not exist
- 8 Illegal symbol in number
- 9 Illegal delimiter
- 10 Invalid position specified

Class 7 – JCL interpreter and trapping errors

Code

- 1 Command line not recognised
- 2 Command name unknown
- 3 Illegal type of parameter
- 4 Unable to change command stream (via IN command)
- 5 Illegal set trap attempt

Class 8 – Programming language run time errors

Code

- 1 Unable to find C program
- 4 Too much heap used
- 5 Create bad pointer
- 6 Heap corrupted
- 7 Dispose nil pointer
- 8 Dispose bad pointer
- 9 Set element out of range
- 10 Parameter too large for EXP
- 11 Negative parameter with SQRT
- 12 Invalid exponentiation (**)
- 13 Invalid parameters with ATAN2
- 14 Negative parameter with LOG
- 15 Invalid parameter with ASIN
- 16 Invalid parameter with ACOS
- 111 FORTRAN Assigned GOTO fault.
Label not in specified list, or not defined.

Class 9 JCL command errors

Code

- 1 Faulty compilation
- 10 Unable to open library
- 11 Maximum number of libraries already open
- 12 Unable to delete library
- 20 Unable to FLIP file
- 30 Unable to initialise profiling

Index

::, [174](#)
;, [174](#)

ABORT, [217](#)
ADD.PROC, [131](#)
ADD.SEGMENT, [132](#)
ADD.TYPE, [132](#)
ALT.ST, [209](#)

BLOCK, [169](#)
BNF, [161](#)
BREAK.INPUT [BI](#), [76](#)
BREAK.OUTPUT [BO](#), [80](#)

CANT.DO, [156](#), [324](#)
CAPTION [CAP](#), [87](#)
CATALOGUE.FILES, [95](#)
CATALOGUE.PERMIT, [97](#)
CATALOGUE.PROCESSES, [304](#)
CATALOGUE.USERS, [149](#)
CHANGE.ACCESS, [296](#)
CHANGE.CHANNEL, [314](#)
CHANGE.COMMON.SEGMENT, [297](#)
CHANGE.DEST, [66](#)
CHANGE.FILE.SIZE, [100](#)
CHANGE.ROOT.DIR, [99](#)
CHANGE.SIZE, [295](#)
CHAR.CONST, [176](#)
CHAR.STRING, [177](#)
CHECK.POINT, [223](#)
CHECK.WAIT, [224](#)
CLEAR.LINE, [123](#)
CLEAR.SCREEN, [123](#)
CLOSE.DIR, [133](#)
CMAP, [179](#)
CODE, [178](#)
COMPARE, [103](#)
COMPARISON, [207](#)
COMPUTATION, [202](#)

CONDITION, [207](#)
CONFIG, [142](#), [145](#)
CONNECT, [155](#)
CONST, [174](#)
COPY.BLOCK, [297](#)
COPY.FILE, [101](#)
COPY.TAPE, [329](#)
CREATE.FILE.SEGMENT, [99](#)
CREATE.FILE.X.SEGMENT, [99](#)
CREATE.PROCESS, [302](#)
CREATE.SEGMENT, [294](#)
CREATE.SEMAPHORE, [319](#)
CREATE.SUBDIR, [99](#)
CREATE.TASK, [306](#)
CREATE.USER, [148](#)
CREATE.X.SEGMENT, [295](#)
CURRENT.INPUT, [76](#)
CURRENT.OUTPUT, [79](#)
CURRENT.TASK, [306](#)

DATAVEC, [187](#)
DEC.INTEGER, [175](#)
DECLARATIVE.STATEMENT, [181](#)
DEFINE.FONT, [281](#)
DEFINE.INPUT [DI](#), [62](#)
DEFINE.IO [DIO](#), [65](#)
DEFINE.OUTPUT, [63](#)
DEFINE.STRING.IO, [66](#)
DELETE [DEL](#), [100](#)
DELETE.FILE, [95](#)
DELETE.SUBDIR, [99](#)
DELETE.TASK, [306](#)
DELETE.USER, [151](#)
DESPool, [157](#)
DIRECTIVE, [177](#)
DIRECTIVE.STATEMENT, [178](#)
DRAW, [254](#)

ECHO.LINE, [89](#)

- ED.READ, [325](#)
- ED.WRITE, [325](#)
- EDIT, [102](#), [105](#)
- EDIT.DISPLAY, [125](#)
- END.INPUT [EI](#), [67](#)
- END.OUTPUT [EO](#), [67](#)
- END.POS, [81](#)
- ENTER.TRAP, [216](#)
- ERASE.USER, [149](#)
- EXAMINE [E](#), [227](#)
- EXPORTS, [165](#)

- FILE [FL](#), [95](#)
- FILMAN, [103](#)
- FIND.ALL.RDS, [133](#)
- FIND.FILE, [328](#)
- FIND.FILES [FE](#), [101](#)
- FIND.LIB.NAME, [133](#)
- FIND.PROC, [134](#)
- FIND.RD, [133](#)
- FIND.SEG.RD, [133](#)
- FIND.U, [150](#)
- FINDF, [135](#)
- FINDN, [134](#)
- FINDP, [134](#)
- FINDTD, [135](#)
- FLIP, [253](#)
- FLOPPY, [104](#)
- FONT.ENQ, [286](#)
- FORCE.INT, [305](#)
- FORMAT [F](#), [229](#)
- FREE.PROCESS, [303](#)

- GET.SEG, [131](#)
- GKS, [259](#)
- GLOBAL, [178](#)
- GO.ST, [208](#)
- GRA.CLIP, [287](#)
- GRA.CMD, [282](#)
- GRA.TEXT, [285](#)

- GRA.TEXT.EXTENT, [285](#)

- HEX.CONST, [175](#)
- HEX.DIGIT, [175](#)
- HEX.DIGITS, [175](#)
- HEX.SEQUENCE, [175](#)

- I.DOC, [78](#)
- I.ENQ, [78](#)
- I.MODE, [76](#)
- I.POS, [77](#)
- I.REC, [77](#)
- I.SEG, [76](#)
- I.SIZE, [77](#)
- I.SOURCE, [77](#)
- IBM, [104](#)
- IMPERATIVE.STATEMENT, [201](#)
- IMPORT, [192](#)
- IMPORTS, [165](#)
- IN, [47](#)
- IN.BACKSPACE, [83](#)
- IN.BIN, [83](#)
- IN.C.LIT, [86](#)
- IN.C.STR, [86](#)
- IN.CH, [82](#)
- IN.GRA.TEXT, [286](#)
- IN.HEX, [86](#)
- IN.I, [85](#)
- IN.LINE, [83](#)
- IN.NAME, [86](#)
- IN.OCT, [86](#)
- IN.REAL, [86](#)
- IN.REC, [85](#)
- IN.STR, [87](#)
- IN.VEC, [83](#)
- INFORM, [179](#)
- INIT, [180](#)
- INIT.IO, [62](#)
- INIT.RD, [234](#)
- INIT.SEMAPHORE, [319](#)

INT.JOB.JOB, [154](#)
INTERCHANGE, [296](#)

KILL, [48](#), [302](#)

LABEL.DEC, [182](#)
LABEL.DISCS, [157](#)
LANGUAGE, [43](#)
LIBRARY, [45](#)
LIBRARY LIB, [45](#)
LINE, [281](#)
LINK, [136](#)
LIST.ACCOUNTS, [152](#)
LIST.DICT, [246](#)
LIST.DIR, [101](#)
LIST.FILE LF, [101](#)
LIST.INDEX, [246](#)
LIST.JOBS, [157](#)
LIST.MOD, [246](#)
LIST.PAGE.PROFILE, [226](#)
LIST.PERMIT, [102](#)
LIST.PROCS LP, [227](#)
LIST.PROFILE, [226](#)
LIT.DEC, [185](#)
LITERALS, [187](#)
LOAD.PROG, [45](#)
LOOK.UP.N, [135](#)
LOOK.UP.PROCESS, [303](#)

MAP, [297](#)
MARKERS, [287](#)
MC68000, [401](#)
MC68010, [297](#), [401](#)
MERGE.DIR, [133](#)
MODIFY M, [229](#)
MODULE, [164](#)
MOUNT, [324](#)
MT.READ, [325](#)
MT.REWIND, [326](#)
MT.SKIP, [325](#)
MT.SKIP.TM, [326](#)

MT.WRITE, [325](#)
MT.WRITE.TM, [326](#)
MU6, [147](#)
MUSL, [161](#)
MUSS, [1](#)
MVEF, [200](#)

NAME, [174](#)
NAME.DIR, [98](#)
NAME.LIST, [165](#), [182](#)
NEW.FONT, [287](#)
NEW.LINES NL, [87](#)
NEW.PASSWORD, [149](#)
NEW.USER, [151](#)
NEXT.CH, [82](#)
NUMERICTYPE, [182](#)

O.DOC, [80](#)
O.MODE, [79](#)
O.POS, [80](#)
O.REC, [79](#)
O.SEG, [79](#)
O.SIZE, [80](#)
OLD, [100](#)
OPEN.DIR OD, [94](#)
OPEN.FILE, [95](#)
OPERAND, [193](#)
OPERATORS, [202](#)
OUT.BACKSPACE, [84](#)
OUT.BIN, [84](#)
OUT.CH, [84](#)
OUT.CHECK.POINT, [224](#)
OUT.CODE, [231](#)
OUT.DATE, [88](#)
OUT.FN, [89](#)
OUT.HDR, [81](#)
OUT.HEX, [88](#)
OUT.I, [87](#)
OUT.LINE, [84](#)
OUT.LINE.NO, [89](#)

- OUT.M, [218](#)
- OUT.MACHINE, [89](#)
- OUT.NAME, [89](#)
- OUT.O.STACK, [231](#)
- OUT.PROG_n OP_n, [231](#)
- OUT.REAL, [88](#)
- OUT.REC, [85](#)
- OUT.STACK OS, [231](#)
- OUT.STATS, [154](#)
- OUT.T, [218](#)
- OUT.TD, [88](#)
- OUT.TIME, [88](#)
- OUT.VEC, [85](#)
- OUTPUT.FONT, [287](#)

- P.SEMAPHORE, [319](#)
- PART, [198](#)
- PASS.SEGMENT, [298](#)
- PASS.SEMAPHORE, [319](#)
- PAYOUT, [150](#)
- PDP11, [172](#), [293](#), [297](#)
- PERMIT, [97](#)
- POSITION.CURSOR, [123](#)
- POSN, [199](#)
- POST.MORTEM, [227](#)
- PREP.QUENU, [124](#)
- PROC.DEC, [184](#)
- PROC.DEFN, [169](#)
- PROC.HEADING, [169](#)
- PROCESS.NAME, [68](#)
- PROFILE, [225](#)
- PROMPT, [89](#)
- PROMPT.CH, [90](#)
- PSPEC, [184](#)

- QUENU, [124](#)
- QUIT, [234](#)

- R.SIZE, [77](#)
- RE.TRAP, [217](#)
- READ.CH, [315](#)
- READ.CH.STATUS, [314](#)
- READ.CHAR, [123](#)
- READ.FILE, [328](#)
- READ.FILE.STATUS, [96](#)
- READ.INT.TRAP, [215](#)
- READ.IO.STATUS, [145](#)
- READ.MACHINE.STATUS, [157](#)
- READ.MESSAGE, [312](#)
- READ.PROCESS.STATUS, [303](#)
- READ.RECOVERY.STATUS, [216](#)
- READ.SEGMENT.STATUS, [296](#)
- READ.STATS, [153](#)
- READ.TAPE, [328](#)
- READ.TIMER, [304](#)
- REAL.CONST, [177](#)
- References, [vi](#)
- RELEASE, [324](#)
- RELEASE.DIR.NAME, [99](#)
- RELEASE.LIB RL, [47](#)
- RELEASE.SEGMENT, [295](#)
- REMOVE.CHECK.POINT, [224](#)
- REMOVE.TRAPS, [216](#)
- RENAME, [100](#)
- RENAME.FILE, [95](#)
- RETURN R, [234](#)
- RETURN.FROM.TRAP, [217](#)
- RUN, [45](#), [47](#)

- SAVE, [100](#)
- SCALARTYPE, [182](#)
- SECURE.SEGMENT, [297](#)
- SELECT.DATA SD, [230](#)
- SELECT.DOC, [76](#)
- SELECT.GRA, [281](#)
- SELECT.HEADER, [76](#)
- SELECT.INPUT SI, [75](#)
- SELECT.OUTPUT SO, [79](#)
- SELECT.SOURCE, [225](#)
- SEND.MESSAGE, [311](#)
- SET.ACCOUNT, [151](#)

SET.CH.STATUS, 312
SET.COMMON.SEGMENT, 145
SET.CONNECTION, 316
SET.I.POS, 82
SET.I.REC, 81
SET.INT.TRAP, 215
SET.NETWORK, 141
SET.O.POS, 82
SET.O.REC, 81
SET.PERIPHERAL, 142
SET.RECOVERY.STATUS, 216
SET.SUPERVISOR, 142
SET.TIME.AND.DATE, 156
SET.TIMER, 304
SET.TRAP, 216
SET.USER.PARAM, 149
SET.VDU.MODE, 122
SIZE, 182
SKIP.LINE, 83
SPACE.DEC, 190
SPACES SP, 87
STEER.PROFILE, 226
STOP, 47
STOPC, 179
SUSPEND.PROCESS, 303
SYSTEM.STATS, 153

TASK, 306
TEMPORARY.LABEL, 156
TERMINATE.PROCESS, 302
TIME.AND.DATE, 156
TL, 398
TL.ASS, 358
TL.ASS.ADV, 360
TL.ASS.END, 360
TL.ASS.VALUE, 359
TL.BLOCK, 371
TL.BOUNDS, 405
TL.C.LIT.128, 360
TL.C.LIT.16, 360
TL.C.LIT.32, 360
TL.C.LIT.64, 360
TL.C.LIT.S, 361
TL.C.NULL, 361
TL.C.TYPE, 361
TL.CALC, 363
TL.CHECK, 394
TL.CODE.AREA, 350
TL.COMMON, 351
TL.CUR.SOURCE, 405
TL.CV.CYCLE, 396
TL.CV.LIMIT, 397
TL.CYCLE, 395
TL.D.TYPE, 393
TL.DATA.AREA, 350
TL.END, 400
TL.END.BLOCK, 371
TL.END.COMMON, 352
TL.END.MODULE, 400
TL.END.PROC, 370
TL.ENQ, 401
TL.ENQ.REG, 402
TL.ENTRY, 370
TL.EQUIV.POS, 352
TL.GLOBAL, 400
TL.I.PARAM, 370
TL.INSERT, 395
TL.LABEL, 372
TL.LABEL.SPEC, 372
TL.LINE, 404
TL.LIT, 362
TL.LOAD, 163, 350
TL.MAKE, 356
TL.MODE, 400
TL.MODULE, 400
TL.PARAM.NAME, 369
TL.PIC, 373
TL.PL, 373
TL.PRINT, 406
TL.PROC, 369

TL.PROC.KIND, [369](#)
TL.PROC.PARAM, [368](#)
TL.PROC.RESULT, [369](#)
TL.PROC.SPEC, [366](#)
TL.RANGE, [395](#)
TL.REG, [397](#)
TL.REPEAT, [396](#)
TL.S.DECL, [353](#)
TL.SEG, [162](#), [349](#)
TL.SELECT.FIELD, [357](#)
TL.SELECT.VAR, [356](#)
TL.SET.TYPE, [357](#)
TL.SOURCE, [404](#)
TL.SPACE, [352](#)
TL.TYPE, [345](#)
TL.V.DECL, [354](#)
TLLOAD, [180](#)
TLMODE, [180](#)
TLSEG, [180](#)
TXT, [237](#)
TYPE, [182](#)
TYPE.DEC, [188](#)

V.SEMAPHORE, [320](#)
V.STORE.DEC, [191](#)
VAR.DEC, [182](#)
VAX, [147](#), [297](#)
VDU.MODE, [123](#)
VDU.NUMBER, [121](#)
VDU.SIZE, [122](#)
VDU.TYPE, [121](#)
VSTORE, [181](#)
VTYPE, [181](#)

WAIT, [313](#)
WIND, [82](#)
WRITE.FILE, [327](#)
WRITE.LABEL, [324](#)
WRITE.TAPE, [327](#)